Scheduling and Rostering with Learning Constraint Solvers

Nicholas Ronald Downing
orcid.org/0000-0002-5357-6586

Submitted in fulfilment of the requirements
for the degree of Doctor of Philosophy

10/2016

Department of Computing and Information Systems
The University of Melbourne

Sponsor: Data61 CSIRO
Advisors: Prof Peter J. Stuckey, Dr Thibaut Feydy
Committee: Prof Mark Wallace, A/Prof Harald Søndergaard

Abstract

In this research we investigate using Constraint Programming (CP) with Lazy Clause Generation (LCG), that is, constraint solvers with nogood learning, to tackle a number of well known scheduling, rostering, and planning problems. We extend a number of CP constraints to be useable in an LCG solver, and we investigate alternative search strategies that use the Unsatisfiable Cores, i.e. reasons for failure returned by the LCG solver, to guide the search for optimal solutions. We give comprehensive analysis and experiments. These experiments show that while adding more and more sophisticated constraint propagators to LCG delivers diminishing returns, unsatisfiable-core optimization which leverages the infrastructure provided by LCG can deliver significant benefits which are unavailable in CP without LCG. Overall, we demonstrate that LCG is a highly competitive technology for solving realistic industrial scheduling and rostering problems to optimality, allowing that the problems are smaller than those tackled by competing algorithms which do not prove optimality.

Preface

This research was carried out in collaboration with my supervisors Prof. Peter J. Stuckey and Dr. Thibaut Feydy. We regularly met to discuss progress and future directions. I co-authored the following conference papers and technical reports with my supervisors:

- Downing, N., Feydy, T., Stuckey, P.J.: Explaining alldifferent. In: *Proceedings of the Thirty-fifth Australasian Computer Science Conference - Volume 122*, *ACSC '12*, pp. 115–124. Australian Computer Society, Inc., Darlinghurst, Australia, Australia (2012)
- Downing, N., Feydy, T., Stuckey, P.J.: Explaining flow-based propagation. In: N. Beldiceanu, N. Jussien, É. Pinson (eds.) *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems: 9th International Conference, CPAIOR 2012, Nantes, France, May 28 – June 1, 2012. Proceedings*, pp. 146–162. Springer Berlin Heidelberg (2012)
- Downing, N., Feydy, T., Stuckey, P.J.: Unsatisfiable cores for constraint programming. *CoRR abs/1305.1690* (2013). URL `http://arxiv.org/abs/1305.1690`
- Downing, N., Feydy, T., Stuckey, P.J.: Unsatisfiable cores and lower bounding for constraint programming. *CoRR abs/1508.06096* (2015). URL `http://arxiv.org/abs/1508.06096`

I received some hands-on editorial assistance with the above papers. The thesis versions are considerably evolved from the original work and contain 100% new material which I created, that is, new background, analysis, examples, implementation, and experiments. All writing presented here is my own work. My supervisors proof-read the written work and suggested corrections and clarifications. Existing research is duly cited and credited.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Combinatorial Optimization is a field of mathematics and computer science in which we try to find an optimal object from a finite set of objects. In what follows we refers to such objects as *solutions* of a Combinatorial Optimization problem. Combinatorial Optimization arises in industrial settings, such as finding the best schedule for an activity.

Combinatorial Optimization is very difficult because the solution set is combinatorial, that is it grows exponentially with the number of choices that can be made to construct a solution. The description of allowable solutions and the optimization criteria can be arbitrarily complex, so there may be no general way to derive a solution other than search.

Constraint Programming (CP) is a declarative problem solving framework, which is a relatively recent development, and which focuses on solving Combinatorial Optimization problems. In CP, problems are specified using variables, subject to constraints.

Combinatorial Optimization problems are typically modelled in CP using constraints including, but not limited to,

- implications or logical constraints – reasoning about truth values; or
- arithmetic or linear constraints – reasoning about numeric values; or
- other primitive constraints – taking the maximum of a collection of variables, looking up a variable from an array via an index variable, etc; or
- global constraints – specifying some interesting property over a collection of variables, e.g. *alldifferent* states all variables in the collection must take different values.

The constraints in a Combinatorial Optimization problem modelled in CP typically express relationships between the following kinds of variables,

- Boolean – commonly seen in Boolean Satisfiability (SAT), Maximum Satisfiability (MaxSAT), and Pseudo-Boolean (PB) solving; or
- Integer – commonly seen in Finite Domain (FD) solving; or
- Real – commonly seen in Linear Programming (LP) solving.

In this research, we will tackle realistic industrial problems using these kinds of constraints and variables, in particular FD problems, but we will also use SAT and LP methods.

1.1 Learning constraint solving

CP solving is highly combinatorial, and typically we do not have closed form solutions or iterative methods as seen in Mathematical or Linear Programming (MP/LP), except on certain kinds of subproblems or subsets of the constraints.

Hence CP solving normally uses a Davis-Putnam-Loveland-Logeman (DPLL) based algorithm [32, 33], which means an exhaustive search for solutions, using filtering to eliminate non-solutions and thereby avoid some of the search cost.

Global constraints have been intensively studied since the introduction of CP [12]. Through a deep understanding of the global properties of the constraint, we can implement *propagation* (that is, filtering) algorithms for specific global constraints, to enhance the performance of the DPLL algorithm by avoiding more of the search cost.

Initially, global propagators were the only reasonable way to reduce search in CP. More recently, clause learning as used in SAT, equivalently *nogood learning*, has emerged as another very powerful way to reduce search in CP [36, 60, 61, 107].

Ohrimenko et al. proposed Lazy Clause Generation (LCG) [83], in which a constraint problem is gradually encoded into clauses, allowing a learning SAT solver such as *GRASP* [73, 74], to employ nogoods and reasoning to reduce search.

In LCG, propagators for specific constraints have to be extended to *explain* themselves by clauses, that is they must expose their reasoning about the domains of variables, so that the LCG solver can analyze conflicts discovered by this reasoning.

LCG is an active area of research, with the initial research focusing on adding global constraints or decompositions to LCG [47, 49, 50, 99, 101], so that LCG solvers could have at least the capabilities of current CP solvers, in addition to learning. This research is ongoing [42, 98], but has reached a useful state – current LCG solvers are very capable. Other research focuses on applying LCG to specific problems [100, 102].

More and more powerful LCG solvers have become available, such as the G12 *LazyFD* solver [41], *Chuffed* [22] and *CPX* [45]. For this research we developed a prototype LCG solver called *LCG-glucose*, which was the fastest solver in the 2016 MiniZinc Challenge in the FD and free search categories [30]. The next fastest was *Chuffed*, demonstrating that LCG is the best current way to solve many highly combinatorial CP problems.

1.2 Constraint based optimization

In its simplest form, CP is just finding solutions that satisfy constraints. Realistic industrial problems are usually optimization problems not satisfaction problems, yet they are highly combinatorial, so we would like to solve them using CP, as opposed to MP/LP methods which are good at optimization but less good at highly combinatorial problems.

Traditionally, optimization in CP uses a `LSSU` (Linear Search Sat/Unsat) algorithm [76], or equivalently, a branch-and-bound approach, in which we treat the problem as a satisfaction problem, find some solution, and then look for incrementally better solutions. This is because CP normally does not 'know' where the good solutions are, unlike MP/LP methods which tackle optimization problems by iteratively improving the objective.

Other technologies related to CP and LCG, such as Maximum Satisfiability (MaxSAT), Answer Set Programming (ASP) and Satisfiability Modulo Theory (SMT), use Unsatisfiable Core approaches to optimization [5, 6, 18, 48, 69, 71, 72, 77, 79]. This is a dual approach where the solution is initially assumed to be optimal despite this violating some constraints, and gradually the violations are addressed in order to approach feasibility.

Since LCG solvers can explain themselves by clauses, they can also reason about unsatisfiability in CP. LCG solvers already prove a problem infeasible by deriving a set of constraints which cannot hold simultaneously, that is, an unsatisfiable core. If these constraints include artificial 'optimality' constraints, then we can see the reasons why a solution cannot be optimal, and we can very selectively remove or relax such constraints.

1.3 Our research questions

In this research we consider using Constraint Programming with learning, in particular the LCG approach pioneered by Ohrimenko et al. [83], to solve realistic industrial scheduling and rostering problems. We focus on problems expressible with constraints over integer variables, that is FD solving, while taking elements from SAT and LP solving.

Given that many of the example problems solved with CP and LCG solvers are essentially puzzles (a few of which we look at in our initial experiments in Chapter 3), we set out to encode some *large, realistic* industrial problems into CP and solve with LCG, and to develop the LCG solver in whatever direction necessary to solve such problems.

Our research question was essentially how far can this approach be taken. Given typical industrial scheduling and rostering problems, which are well described in the literature, how large an instance can we tackle, and what can we do to improve this limit? In this goal we focus mainly on exact-solution (proveably optimal) approaches, as is natural for CP, so our instances will begin somewhat smaller than other researchers.

Adding constraints to LCG, by extending them to explain themselves, has been very helpful for specific problems such as RCPSP/max [102] which uses the *cumulative* constraint with explanations [99]. We followed such an approach, but as the research progressed we found that this approach, while helpful, was not decisive on our problems.

As well as adding constraints to LCG, more recent research focuses on innovative uses of LCG which are not applicable to traditional CP solvers [23, 24, 25]. As the research progressed we focused more and more on this approach, so the research question became, how can we leverage the infrastructure provided by LCG to solve these problems.

1.4 Our research contributions

In the initial part of this thesis, we examine several constraints which are useful for scheduling, rostering and planning, and extend their propagation algorithms to have explanation capability so that they can be integrated into a state-of-the-art LCG solver. Earlier versions of these chapters appeared in ACSC '12 and CPAIOR '12:

- Downing, N., Feydy, T., Stuckey, P.J.: Explaining alldifferent. In: *Proceedings of the Thirty-fifth Australasian Computer Science Conference - Volume 122*, *ACSC '12*, pp. 115–124. Australian Computer Society, Inc., Darlinghurst, Australia, Australia (2012)
- Downing, N., Feydy, T., Stuckey, P.J.: Explaining flow-based propagation. In: N. Beldiceanu, N. Jussien, É. Pinson (eds.) *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems: 9th International Conference, CPAIOR 2012, Nantes, France, May 28 – June 1, 2012. Proceedings*, pp. 146–162. Springer Berlin Heidelberg (2012)

We describe for the first time the learning version of the bounds($\mathcal{Z}$)-consistent *alldifferent* propagator [65, 88]. As a result we can give state-of-the-art results on the Superblock Instruction Scheduling problems [68] used in evaluating the previous work.

We give the most comprehensive theoretical analysis to date of the learning versions of the global constraint propagators using matching [61, 90, 95] or flow theory [2, 19, 31, 91, 94, 106], greatly expanding upon the existing literature. As a result we can give state-of-the-art results on the Personnel Scheduling problems [19, 106] used in the previous work.

We give comprehensive experiments to show that LCG solvers should definitely offer these constraints for use on specific problems, but also illustrating that the tradeoffs for propagation are quite different in LCG solvers than traditional CP solvers. This prompts us to rethink the need for more and more global propagators in LCG, since LCG solvers are to some extent capable of deriving globality through learning.

In the latter part of this thesis, we examine several important unsatisfiable-core solving methods in other technologies such as Maximum Satisfiability (MaxSAT), Answer Set

Programming (ASP) or Satisfiability Modulo Theories (SMT), extend them to allow an integer linear objective so that they can tackle CP optimization problems, and integrate them into our state-of-the-art LCG solver, using LCG to generate unsatisfiable cores.

Our initial efforts focused on soft-constraint problems in CP using unsatisfiable cores with a Pseudo-Boolean (PB) objective, which we describe in technical reports:

- Downing, N., Feydy, T., Stuckey, P.J.: Unsatisfiable cores for constraint programming. *CoRR abs/1305.1690* (2013). URL `http://arxiv.org/abs/1305.1690`
- Downing, N., Feydy, T., Stuckey, P.J.: Unsatisfiable cores and lower bounding for constraint programming. *CoRR abs/1508.06096* (2015). URL `http://arxiv.org/abs/1508.06096`

The second report includes our early attempts to consider a broader range of problems (eventually leading to the version in this thesis), and gives some novel extensions.

We give the most comprehensive theoretical analysis to date of the MaxSAT unsatisfiable-core algorithms, showing similarities and differences that were masked by the original authors' widely different presentation or implementation of the same ideas. As a result of this analysis, we are able to extend all of the algorithms considered, to CP.

We examine a suite of existing industrial scheduling and rostering problems [37, 105, 113, 114], that benefit from unsatisfiable cores, and demonstrate that LCG solvers should definitely offer an unsatisfiable-core optimization mode for use on suitable problems.

We entered an *LCG-glucose* variant using our CP extension of the `OLL` unsatisfiable-core algorithm [77], in the free search category of the 2016 MiniZinc Challenge [30].

On 6 of the 17 optimization problems in the challenge, *LCG-glucose* with `OLL` was faster than *LCG-glucose* with `LSSU`, the standard variant. On 4 of these problems *LCG-glucose* with `OLL` was the fastest overall. On the remaining 2 problems *Chuffed* was the fastest, which is similar to *LCG-glucose* and could also be extended to use our `OLL` algorithm.

Overall, we demonstrate that LCG is a highly competitive technology for solving realistic industrial scheduling and rostering problems to optimality, allowing that the problems are smaller than those tackled by competing algorithms which do not prove optimality.

Chapter 2

Background

2.1 Mathematical notation

The algorithms given in this thesis, and their associated definitions, theorems, etc, will be quite technical, so we will begin by establishing some essential notation.

The most common data structures that we will use for explanatory purposes will be sets, arrays (lists), dictionaries (maps), tuples (pairs, etc), and Haskell-like discriminated unions (or equivalently, Prolog-like uninterpreted function symbols).

Algorithms will be expressed in an informal pseudocode. We use common control-flow constructs like '**if** ... **then**', with indentation to indicate the scope of the control-flow constructs or blocks. We use C-like '**function**' semantics. '#' is a comment.

Multiple-letter variable or function names will be set in italics like '*var*', as is conventional for simple names like i, j, x, y. We also set all other identifiers, such as discriminated union or map keys, in italics. The intended meaning should be clear from the context.

For sets we use the standard mathematical notation $\{element, element, \ldots\}$, where $\emptyset$ denotes the empty set, with operators '$\in$' for membership, '$\cup$' for union, '$\cap$' for intersection, '$\setminus$' for deletion, '$|\ldots|$' for cardinality, and '$\{element : predicate\}$' for comprehensions. (We enclose mathematical operators in single quotes for discussion purposes).

For lists we use a Python-like syntax $[element, element, \ldots]$ for an ordered list with indices $1, 2, \ldots$. Given var is a list, its length is $|var|$ and its elements are $var[i]$ where $i \in 1..|var|$. We use '$\in$' for membership, '+' for concatenation and '$[a..b]$' for slicing.

For dictionaries we use a set notation $\{key \mapsto value, key \mapsto value, \ldots\}$. Dictionaries are functions, given var is a dictionary, $var(k)$ means the value v such that $(k \mapsto v) \in var$. We use '$\cdot$' as a wildcard so that e.g. $\{k : (k \mapsto \cdot) \in var\}$ is the set of keys in var. We also write $var(k) \leftarrow v$ for element replacement, it means $var \leftarrow var \setminus \{k \mapsto \cdot\} \cup \{k \mapsto v\}$.

For discriminated unions we use a keyword like *unsatisfiable*, for example as a status return from a function, or else we use a keyword plus arguments like *satisfiable*(S) which indicates a status and encodes some further information which depends on the status. We unpack these either implicitly or using a Haskell-like '**case** ... **in**' construct.

We write tuples in parentheses like (a, b) and manipulate them similarly to the above, or we use convenience functions like $fst((a, b)) = a$ and $snd((a, b)) = b$.

For Boolean reasoning we use variables with ordinary names like a, b which can be assigned *true* or *false*. We treat 1 interchangeably with *true* and 0 with *false*, so that e.g. $a+b \leq 1$ means at most one of a, b can be assigned *true*. We use '$\vee$' for disjunction, '$\wedge$' for conjunction, '$\neg$' for negation and '$\rightarrow$' for implication, where $A \rightarrow B \equiv \neg A \vee B$.

Boolean expressions appearing in parentheses indicate the truth value of an expression, for instance $(x \geq 5)$ means 0 if $x < 5$ or 1 if $x \geq 5$. For example, $\sum_{i=1}^{10}(x_i \geq 5)$ calculates the number of elements in $x_1, \ldots, x_{10}$ which are ≥ 5. On the other hand, a Boolean expression in square brackets indicates an ordinary Boolean variable with a semantic name, e.g. $[x \geq 5]$ means: the Boolean variable whose name is 'x is greater than or equal to 5'.

We use standard symbols like $\mathbb{Z}$ for the set of integers or $\mathbb{R}$ for the set of reals, occasionally $\mathcal{Z}$ or $\mathcal{R}$ in referencing other sources. We make frequent use of integer-valued intervals, so for brevity we define a custom notation $a..b$ as shorthand for $\{a, a + 1, \ldots, b\}$.

2.2 Boolean Satisfiability (SAT)

SAT solving is, given a collection of Boolean variables $x_1, \ldots, x_n$ and a formula, find a valuation to the variables such that the formula is *true*. Typically the formula is expressed as a conjunction $C_1 \wedge \ldots \wedge C_m$ of *clauses* where each clause C_i is a disjunction $\ell_1 \vee \ldots \vee \ell_k$ of *literals*. A literal is defined as a variable x_i or its negation $\neg x_i$, $i \in 1..n$.

Boolean expressions written as a conjunction of clauses, each of which is a disjunction of literals, are in Conjunctive Normal Form (CNF). One reason why SAT solvers are useful is that they can tackle an arbitrary Boolean expression, after putting it in CNF by term expansion and/or introducing intermediate variables for subexpressions. We can also encode constraints, e.g. linear constraints, into CNF [1, 107]. We will write *CNF*(...) to mean some reasonable CNF encoding of an arbitrary expression or constraint.

Modern SAT solving uses a Conflict-Driven Clause Learning (CDCL) procedure. Earlier SAT solvers had used a DPLL [32, 33] procedure which is an exhaustive backtracking search. In SAT with DPLL, each choicepoint results in 2 branches, the left branch is explored first, and then the right branch is explored after backtracking out of the left branch. CDCL is a subtle but significant modification which only ever explores the left branch, since clause learning cuts off useless branches. The key elements of CDCL SAT solving are:

- *Partial assignment:* Variables $x_1, \ldots, x_n$ are initially marked 'undefined' and progressively given values 0 (*false*) or 1 (*true*) to construct a working partial solution.

Algorithm 2.1 Example of a Boolean Satisfiability (SAT) solving algorithm using CDCL

inputs:

$D_{orig} = \{x \mapsto unset : x \text{ is a variable}\}$
$C = \{C_1, \ldots, C_m\}$ where C_i = a set $\{\ell_1, \ldots, \ell_k\}$ representing a clause $\ell_1 \vee \ldots \vee \ell_k$, where any literal ℓ is a variable x or its negation $\neg x$
A = a set of assumptions as literals to be assigned 1 (*true*) before solving starts

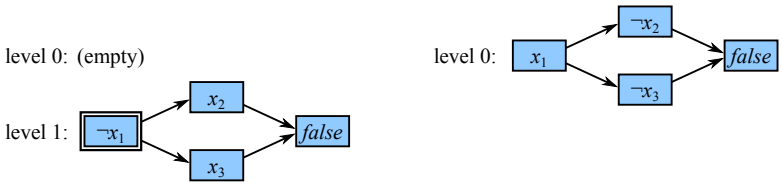
outputs:

satisfiable(S) where S = a map from variable to value satisfying C, or
unsatisfiable(U) where U = a set of assumptions which cannot hold simultaneously

function $SAT(D_{orig}, C, A)$

```
1  D ← D_orig
2  define convenience functions and operators
3      V(x) ≡ 1 if D(x) = 1, 0 if D(x) = 0, unset otherwise,
4      V(¬x) ≡ 0 if D(x) = 1, 1 if D(x) = 0, unset otherwise,
5      V(x) ← 1 ≡ D(x) ← 1,
6      V(¬x) ← 1 ≡ D(x) ← 0
7  level ← 0                                  # current decision level
8  stack ← []                                 # saved assignments per decision level
9  G ← [(assumption, ℓ) : ℓ ∈ A]              # list of assignments for current level
10 for each ℓ ∈ A do V(ℓ) ← 1                  # assign (assume) literals
11 while true do
12     if there is a false or unit clause L ∈ C then
13         if L is false then
14             if level = 0 then return unsatisfiable({¬ℓ : ℓ ∈ analyze_final(L)})
15             L ← analyze(G, L)              # new clause from implication graph
16             C ← C ∪ {L}                    # save new clause in database
17             level ← smallest decision level such that L would be unit
18             (D, G) ← stack[level + 1]      # restore earlier level assignments
19             stack ← stack[1..level]        # backtrack (discard assignments)
20         let {ℓ_1, ..., ℓ_k} = L such that V(ℓ_1) = 0, ..., V(ℓ_{k-1}) = 0, V(ℓ_k) = unset
21         G ← G + [({ℓ_1, ..., ℓ_{k-1}}, ℓ_k)]   # add to implication graph
22         V(ℓ_k) ← 1                         # assign (propagate) literal
23     else if there is an undefined literal (ℓ ↦ unset) ∈ V then
24         level ← level + 1                  # branch (new decision level)
25         stack ← stack + [(D, G)]           # save current level assignments
26         G ← [(decision, ℓ)]                # add to implication graph
27         V(ℓ) ← 1                           # assign (decide) literal
28     else return satisfiable(D)
```

- *Propagation: Unit clauses*, of the form $\ell_1 \vee \ldots \vee \ell_k$ where $\ell_1, \ldots, \ell_{k-1}$ are all *false*, imply that the last literal ℓ_k is *true*. We can write this $\neg\ell_1 \wedge \ldots \wedge \neg\ell_{k-1} \rightarrow \ell_k$.
- *Search:* When propagation reaches a fixed point (no further propagation is possible) the solver *decides* (guesses) a value for a variable before continuing. *Decision levels* are numbered, level 0 means no decisions have been made, level 1 means one decision has



(a) Implication graph at first conflict (b) Implication graph at second conflict

Fig. 2.1 Example implication graphs created by Algorithm 2.1

been made (followed by propagation) and so on. Note that literals *true* at decision level 0 are called *facts*, and will be propagated before making any decisions.

- *Backtracking:* If the current assignment is in *conflict*, i.e. a clause $\ell_1 \vee \ldots \vee \ell_k$ has all literals *false*, the solver backtracks to an earlier level and tries a different assignment.
- *Conflict analysis:* Propagation builds an *implication graph* showing how the current decision (given earlier level assignments) leads to a set of assignments and/or a conflict. Following the implication graph back to a Unique Implication Point (UIP) [73, 74] gives a set of literals that imply the conflict, which is saved as a *learnt clause* to improve future search. Infeasible problems eventually yield the *empty clause* as a learnt clause.

Algorithm 2.1 shows a CDCL SAT solving algorithm. Lines 9–10 and 14 allow assumptions, which we ignore until Section 2.2.3, in the meantime we assume $A = \emptyset$. Lines 12 and 20–22 implement unit clause propagation, or filtering, to keep the current partial assignment V consistent with C. Lines 13–19 implement conflict driven clause learning and backtracking, to record the result of the current search subproblem and resume the search after backtracking. Lines 23–27 implement an arbitrary search algorithm which simply creates a new decision level and assigns any variable. All paths build an implication graph in G and *stack* giving reasons for assignments made, or for search a placeholder reason.

Example 2.1 Consider the SAT problem

$$\text{find } x_1, x_2, x_3 \in 0..1 \text{ such that } C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5 \wedge C_6,$$
$$\text{where } C_1 \equiv \neg x_1 \vee \neg x_2 \qquad C_2 \equiv \neg x_1 \vee \neg x_3 \qquad C_3 \equiv \neg x_2 \vee \neg x_3$$
$$C_4 \equiv x_1 \vee x_2 \qquad C_5 \equiv x_1 \vee x_3 \qquad C_6 \equiv x_2 \vee x_3.$$

Initially no propagation is possible. Search at line 23 detects that $\neg x_1 = \mathit{unset}$ and lines 24–27 set $\neg x_1$. Line 12 detects that C_4 is unit and lines 20–22 set x_2. Similarly, C_5 is unit which sets x_3. Lines 12–13 detect that C_3 is falsified and lines 14–19 derive a *fact* or singleton learnt clause, x_1. Then a similar process follows which propagates C_1 and C_2 before detecting a

Algorithm 2.2 A conflict analysis procedure for a CDCL SAT solver

inputs:

G = list of $(reason, assignment)$ since last choicepoint (implication graph)
L = set of literals $\{\ell_1, \ldots, \ell_k\}$ giving a clause $\ell_1 \vee \ldots \vee \ell_k$ falsified by G

outputs:

set of literals giving a falsified clause which was unit at last choicepoint

function *analyze*(G, L)

```
1  let F = {¬ℓ : (·, ℓ) ∈ G}                    # all literals falsified at this level
2  while |L ∩ F| > 1 do                          # continue until first UIP
3      let (R, ℓ) = G[i], ¬ℓ ∈ L, i maximal      # most recent useful assignment
4      L ← L\{¬ℓ} ∪ R                           # explain literal by its reason
5  return L                                     # report derived (learnt) clause
```

conflict in C_6, falsifying the entire SAT problem. We summarize these steps as follows,

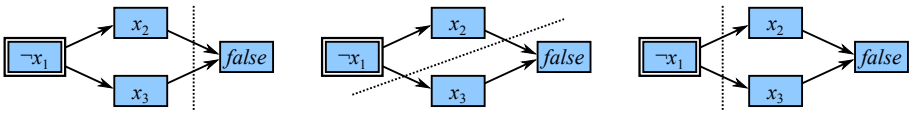
level	assignments G	action
0	[]	$\neg x_1$ is *unset*, decide
1	$[(decision, \neg x_1)]$	C_4 is unit, propagate
1	$[(decision, \neg x_1), (\{x_1\}, x_2)]$	C_5 is unit, propagate
1	$[(decision, \neg x_1), (\{x_1\}, x_2), (\{x_1\}, x_3)]$	C_3 is *false* (see Figure 2.1a), backtrack
0	$[(\emptyset, x_1)]$	C_1 is unit, propagate
0	$[(\emptyset, x_1), (\{\neg x_1\}, \neg x_2)]$	C_2 is unit, propagate
0	$[(\emptyset, x_1), (\{\neg x_1\}, \neg x_2), (\{\neg x_1\}, \neg x_3)]$	C_6 is *false* (see Figure 2.1b), unsatisfiable.

Figures 2.1a and 2.1b show the implication graphs encoded by the stack plus the conflict clause at each conflict. Decision level 0 collects any derived facts and is never backtracked past. Decision levels 1 and above start with a decision, which is shown double-boxed. □

2.2.1 Conflict analysis in SAT

Algorithm 2.2 shows an analysis procedure to derive learnt clauses on behalf of Algorithm 2.1. The subroutine *analyze*() takes a clause which is falsified by the current assignment, and returns a falsified clause which would have been unit at the last choicepoint. Therefore it cuts off the current subproblem. An effective choice of such a clause is the First Unique Implication Point (1UIP), introduced in the learning SAT solver *GRASP* [73].

To calculate the 1UIP, we take G = an ordered list of assignments made since the last choicepoint and L = the initial conflict clause, and progressively resolve a clause from G against L, working in the reverse order of assignment. The resolution rule is $X \vee \boldsymbol{\neg \ell}$ resolves with $Y \vee \boldsymbol{\ell}$ to create a new clause $X \vee Y$, where X and Y are disjunctions of literals. Thus we can eliminate literals falsified by G from the conflict clause L. When only one literal falsified by G remains in L, then L would be unit disregarding the assignments in G.



(a) Initial conflict (b) After one resolution step (c) After two resolution steps

Fig. 2.2 Conflict analysis example showing iterations of Algorithm 2.2

Example 2.2 Consider Example 2.1 when C_3 is falsified resulting in a call to *analyze*(G, L) with $G = [(\mathit{decision}, \neg x_1), (\{x_1\}, x_2), (\{x_1\}, x_3)]$, and $L = \{\neg x_2, \neg x_3\}$ representing the conflict clause $\neg x_2 \vee \neg x_3$. Figure 2.2a shows the implication graph encoded by G and L, with the dotted line showing the frontier isolating the conflict literals in L from the rest of the graph. Initially the isolated portion consists of only the conflict node *false*.

More than one literal of L is falsified by G, so the loop at line 2 executes. Most recently $C_5 \equiv x_1 \vee x_3$ propagated to assign x_3, which falsified $\neg x_3 \in L$. Considering C_5 as a set of literals gives the assignment set $\{x_1, x_3\}$, which is encoded in $G[3] = (\{x_1\}, x_3)$ by separating the assigned literal x_3 from its reason set $\{x_1\}$. Resolving the assignment set $\{x_1, \boldsymbol{x_3}\}$ with the conflict set $L = \{\neg x_2, \boldsymbol{\neg x_3}\}$ gives a new conflict set $L = \{x_1, \neg x_2\}$. This eliminates $\neg x_3$ from L, shown in Figure 2.2b by moving the frontier back past x_3.

More than one literal of the new L is falsified by G, so the loop at line 2 executes again. The most recent is $\neg x_2$, due to the clause $C_4 \equiv x_1 \vee x_2$. Resolving the assignment set $\{x_1, \boldsymbol{x_2}\}$ with $L = \{x_1, \boldsymbol{\neg x_2}\}$ yields $L = \{x_1\}$. This eliminates $\neg x_2$ from L, shown in Figure 2.2c by moving the frontier back past x_2. Now only the single literal $x_1 \in L$ is falsified by G, so the loop at line 2 terminates, returning the revised conflict set $\{x_1\}$.

The learnt clause is x_1, a singleton clause/fact, which cuts off the previous decision $\neg x_1$. Note that in general the 1UIP nogood does not always reverse the previous decision, but it makes *some* propagation which cuts off the previous subproblem. That is, the previous set of propagations started from the previous decision would eventually encounter an obstacle from the 1UIP nogood, which would trigger a fresh conflict analysis. □

The preceding discussion referred to disjunctions of *false* literals, in order to show the connection between learning SAT solving, conflict analysis, and resolution trees or proofs. In our LCG discussions, we will use a more intuitive notation based on conjunctions of *true* literals, e.g. a unit clause $a \vee b \vee c$ which propagates to set c will be written $\neg a \wedge \neg b \rightarrow c$, and a falsified clause $a \vee b \vee c$ will be written $\neg a \wedge \neg b \wedge \neg c \rightarrow \mathit{false}$.

2.2.2 Unsatisfiability proofs in SAT

Although conflict analysis is normally unnecessary at decision level 0 which contains only facts, Algorithm 2.3 defines a simple extension of the conflict analysis procedure to show how SAT solvers prove a problem unsatisfiable by deriving the empty clause.

Algorithm 2.3 An unsatisfiability proof procedure for a CDCL SAT solver

inputs:

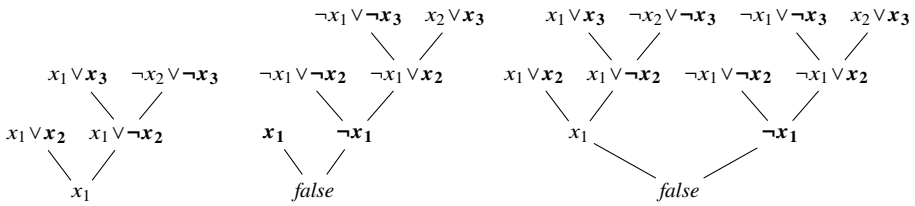
G = list of (*reason*, *assignment*) at decision level 0, must be assumptions or facts
L = set of literals $\{\ell_1, \ldots, \ell_k\}$ giving a clause $\ell_1 \vee \ldots \vee \ell_k$ falsified by G

outputs:

set of literals giving a falsified clause containing only negated assumptions

function *analyze_final*(G, L)

```
1  let F = {¬ℓ : (reason, ℓ) ∈ G, reason ≠ assumption}   # all facts falsified at this level
2  while |L ∩ F| > 0 do                                  # continue until derived false
3      let (R, ℓ) = G[i], ¬ℓ ∈ L, i maximal              # most recent useful assignment
4      L ← L\{¬ℓ} ∪ R                                    # explain literal by its reason
5  return L                                              # report derived (empty) clause
```



(a) Deriving learnt clause x_1 (b) Deriving empty clause (c) Resolution proof of unsatisfiability

Fig. 2.3 Resolution trees produced by conflict analysis

Referring to line 14 of Algorithm 2.1, the SAT solver calls here once unsatisfiability is inevitable. Given we ignore assumptions until the next section, the return value is always $\emptyset$, so this final analysis just 'goes through the motions' to illustrate to the reader, or to a caller who instruments the algorithm to produce a resolution tree, the steps.

The main difference is that the termination condition at line 2 of Algorithm 2.2 changes from '> 1' to '> 0' in this special case. Therefore the conflict analysis continues to explain the literals in L until it derives $\emptyset$ (*false*). The sequence of resolution steps performed to derive $\emptyset$ (*false*) gives what is effectively a formal proof of the unsatisfiability.

Example 2.3 Figure 2.3a shows the resolution tree produced by Example 2.2 to derive the learnt clause x_1. Figure 2.3b shows the resolution tree produced by the final conflict analysis procedure to derive the empty clause. Figure 2.3c shows the entire resolution proof of the unsatisfiability, by replacing the learnt clause x_1 in Figure 2.3b by its derivation. □

2.2.3 SAT solving under assumptions

Most modern SAT solvers support solving under assumptions. An assumption is a variable assignment (or literal) temporarily added to a SAT problem on a particular solving attempt, in order to see what will happen in that scenario. No additional complexity is required, since

the SAT solver can already reason about literals (assumptions or otherwise), we just have to set things up beforehand and report on the assumptions afterwards.

There are several advantages to using assumptions as compared with naively adding the assignments and then using the basic SAT or LCG solver without assumptions. Learnt clauses, including the empty clause, are conditioned by assumptions, so (i) if the problem is unsatisfiable we obtain a set of conflicting assumptions, and (ii) we can remove assumptions without having to delete any learnt clauses derived under assumptions.

Algorithm 2.1 lines 9–10 show the basic SAT solver initializing its implication graph with a set of assumptions, giving a placeholder reason which allows assumptions to be distinguished from facts (i.e. literals which are always *true* as a consequence of the problem clauses). Algorithm 2.3 lines 1–2 show the final conflict analysis procedure deriving an unsatisfiability proof as a clause which is empty *except for negated assumptions*.

Suppose the derived clause is $\neg a_1 \vee \ldots \vee \neg a_n$ where $a_i \in A$ is an assumption, $i \in 1..n$. This means *false* under the assumptions $a_1, \ldots, a_n$. At least one of the negated assumptions $\neg a_1, \ldots, \neg a_n$ must hold, that is, all of the corresponding original assumptions $a_1, \ldots, a_n$ cannot hold simultaneously. We can write this $a_1 \wedge \ldots \wedge a_n \rightarrow \mathit{false}$. Algorithm 2.1 line 14 converts the clause to this latter format, to simplify our later discussions.

Example 2.4 We tackle the problem of Example 2.1 under assumptions $A = \{\neg x_1\}$. Solving proceeds similarly to before, referring to Example 2.1 which had $\neg x_1$ as a decision.

Where previously *analyze*() called at decision level 1 derived $\{x_1\}$ as a learnt clause, here *analyze_final*() called at level 0 derives $\{x_1\}$ as *false* under the assumption $\neg x_1$.

The previous Figure 2.3a gives the resolution tree for this derivation, showing *analyze_final*() deriving a clause which is empty except for the negated assumption x_1.

The return value from *SAT*() is then *unsatisfiable*($\{\neg x_1\}$) giving $\neg x_1$ as an assumption which cannot hold (usually it would be a set which cannot hold simultaneously). □

2.2.4 Autonomous search in SAT

Modern SAT solvers define their own search space, a process referred to as *autonomous search*. Algorithm 2.1 lines 23–27 show the SAT solver choosing an arbitrary literal and assigning it, when the propagation algorithm of lines 12–22 makes no further progress. In practice the choice of literal is not arbitrary but relies on good heuristics to identify promising subproblems. The heuristics are based on statistics collected during search.

An extremely effective heuristic was introduced in the solver *Chaff*, called Variable State Independent Decaying Sum (VSIDS) [78]. In this approach, each Boolean literal receives a counter which is incremented each time the literal is involved in conflict analysis, and decays at some rate. Ranking the undefined literals by this measure makes the search focus on literals that cause conflicts, and hence, on highly constrained subproblems.

A more recent solver *MiniSAT* [40] uses a VSIDS-like approach in which both literals of a variable share the same counter. We write *activity based search* as an umbrella term to describe all versions of the technique, used in most modern SAT solvers.

Another heuristic, for choosing the value of a variable (*true* or *false*) to associate with the current guess, was introduced in the solver *Rsat*, called *phase saving* [85]. In this approach the state of each variable is saved when unfixed by backtracking, and reused the next time it is fixed by search, so as to explore similar subproblems after each backtrack.

SAT solvers sometimes get stuck exploring unpromising subproblems, so the solver usually restarts from the beginning at some periodic rate. Clause learning and phase saving avoid any serious loss of work, and since activities are continually updated, this usually takes the SAT solver straight to a promising subproblem where it can make more progress.

2.3 Constraint Programming (CP)

Constraint Programming (CP) is finding a valuation to some set of variables given *constraints* over those variables. CP is usually tackled using a DPLL algorithm [32, 33], which alternates between *propagation* and *search*. Propagation is filtering unsupported values from the domains of variables. Search is guessing the value of a variable to generate a subproblem, and backtracking if propagation finds a subproblem to be infeasible.

We define integer variables $x_1, \ldots, x_n$ where each variable x_i has a finite integer domain $D_{orig}(x_i) \subset \mathbb{Z}, i \in 1..n$. The Constraint Program consists of a set of constraints $C_1, \ldots, C_m$ where each constraint C_i takes the variables $x_1, \ldots, x_n$ (or some subset) as arguments and defines a set of acceptable tuples over these. We have to find a *model*, as a valuation to $x_1, \ldots, x_n$ where $x_i \in D_{orig}(x_i), i \in 1..n$, such that $C_1, \ldots, C_m$ hold simultaneously.

Example 2.5 Consider x_1, x_2, x_3 where $D_{orig}(x_1) = D_{orig}(x_2) = D_{orig}(x_3) = 1..3$, with constraints $C_1(x_1, x_2, x_3) \equiv x_1 \leq x_2$ and $C_2(x_1, x_2, x_3) \equiv x_2 \leq x_3$. The set of models for this CP is $(v_1, v_2, v_3) \in 1..3^3$ where $v_1 \leq v_2 \leq v_3$. Since each constraint looks at only a subset of variables they can be written $C_1(x_1, x_2) \equiv x_1 \leq x_2$ and $C_2(x_2, x_3) \equiv x_2 \leq x_3$.

To solve these CP problems we can consider subproblems having a restricted domain $D(x_i) \subseteq D_{orig}(x_i)$ for each variable $x_i, i \in 1..n$. For instance we can divide the original CP into some number of subproblems by partitioning $D_{orig}(x_1)$ into $D_1, \ldots, D_k$ which cover $D_{orig}(x_1)$, and then we can set $D(x_1)$ to each of $D_1, \ldots, D_k$ in turn and consider the resulting subproblems. In each subproblem we can further divide $D(x_1)$ or we can consider $D(x_2)$ and so on. This gives a divide-and-conquer strategy for solving CPs.

Given a subproblem as a set of temporary domains $D(x_1), \ldots, D(x_n)$, we can apply propagation (that is, filtering) algorithms. Each constraint $C_i, i \in 1..m$ defines a 'perfect' propagation algorithm P_i which in principle would generate the set K of possible tuples $(v_1, \ldots, v_n) \in D(x_1) \times \ldots \times D(x_n)$ which are acceptable to C_i, failing the subproblem if K

is empty. Then the *supported* values of $x_i, i \in 1..n$ are those which appear in any tuple of K, and the temporary $D(x_i)$ can be replaced with the set of supported values.

The 'perfect' propagation algorithm described above gives *domain consistency* as defined by Van Hentenryck et al. [110, Definition 3.1] or Choi et al. [21, Definition 1].

Definition 2.1 If every value $u \in D(x_i)$ for all $i \in 1..n$ has a *support* as a tuple $(v_1, \ldots, v_n) \in D(x_1) \times \ldots \times D(x_n)$ such that $v_i = u$ and $C_i(v_1, \ldots, v_n)$ holds, then the constraint $C_i, i \in 1..m$ is domain consistent. If all constraints are domain consistent, then the subproblem defined by the domains $D(x_1), \ldots, D(x_n)$ is domain consistent.

Considering only binary constraints, if each value of each variable has a support in the domain of the other variable, it is called *arc consistency*. Then domain consistency is the obvious extension of arc consistency to n-ary constraints. Thus domain consistency can also be called *hyper-arc consistency* or *generalized arc consistency* [21].

Effective CP solving using the traditional DPLL approach requires good heuristics for propagation. For many constraints it is impractical to implement the domain-consistent propagator P_i described in the previous discussion, but good filtering algorithms still exist, which eliminate 'obviously unsupported' values.

A weaker version of consistency is *bounds consistency*, in which we only check the endpoints of $D(x_i)$ for a support. We define $\mathrm{lb}(x_i) = \min D(x_i)$ and $\mathrm{ub}(x_i) = \max D(x_i)$ to facilitate bounds reasoning. In the following definition $a..b$ is the integer-valued interval $\{v \in \mathbb{Z} : a \leq v \leq b\}$ and $[a, b]$ is the real-valued interval $\{v \in \mathbb{R} : a \leq v \leq b\}$.

Definition 2.2 If every endpoint value $u \in \{\mathrm{lb}(x_i), \mathrm{ub}(x_i)\}$ for all $i \in 1..n$ has a *support* as a tuple $(v_1, \ldots, v_n) \in D(x_1) \times \ldots \times D(x_n)$ such that $v_i = u$ and $C_i(v_1, \ldots, v_n)$ holds, then the constraint $C_i, i \in 1..m$ is bounds($\mathcal{D}$) consistent [21, Definition 2]. Relaxing $v_j \in D(x_j), j \neq i$ to $v_j \in \mathrm{lb}(x_j)..\,\mathrm{ub}(x_j)$ defines bounds($\mathcal{Z}$) consistency [21, Definition 3]. Further relaxing this to $v_j \in [\mathrm{lb}(x_j), \mathrm{ub}(x_j)]$ defines bounds($\mathcal{R}$) consistency [21, Definition 4].

In this thesis we mainly consider domain- and bounds($\mathcal{Z}$) or ($\mathcal{R}$)-consistency. Domain consistency gives the strongest filtering and the most reduction in search, but is costly to propagate. Bounds($\mathcal{R}$) consistency gives weaker filtering and less reduction in search, but is usually cheap to propagate. We investigate these tradeoffs in Chapters 3 and 4. There are also some specialized consistency notions for particular constraints.

For efficient solving with traditional CP we also need good heuristics for search. For particular problems we can define an efficient search which helps to move towards a model or fail uninteresting subproblems early. Therefore CP solving usually relies on *programmed search*. The programmed search states the order to consider each variable in generating subproblems, and how the variable's domain should be subdivided.

If the ordering is known in advance it is called *static search*, if the ordering depends on information known during search, such as domain sizes, it is called *dynamic search*.

Dynamic searches are semi-autonomous searches, in the sense that the modeller specifies an interesting set of variables to use for generating subproblems and the solver chooses among those. In this thesis we only use dynamic search, mainly *first_fail* i.e. smallest-domain-first. Traditional CP does not use autonomous search (described in Section 2.2.4).

2.3.1 History of learning in CP

The recent breakthrough in CP solving has been to apply learning solver technology to avoid repeated work, as is taken for granted in modern SAT solvers. How to do this was not immediately obvious, due to the more general CP constraints and variable domains, and their correspondingly more complex implication graphs.

The earliest work in CP-based learning was by Dechter [36] which introduces among other techniques the idea of recording partial solutions that do not lead to a solution, as a list of equalities. These nogoods are very specific (being essentially a list of decisions omitting any that did not matter) and do not prune as well as in later work.

The next important work was by Katsirelos and Bacchus [62] which allows disequalities to appear in a nogood, referred to as a *generalized nogood*. This allows nogoods to express reasoning over domains, e.g. a propagator for $x = y$ can produce nogoods that set y to equal x for specific values of x. They show that generalized nogoods are beneficial.

Another influential work was by Tamura et al. [107] who implemented the simple idea of decomposing a CP problem to SAT and then solving with an unmodified learning SAT backend. In contrast to Katsirelos and Bacchus, Tamura et al. use a bounds (*order*-) encoding of domains rather than (dis)equalities. Both encodings are valuable for specific kinds of constraints. But, Tamura et al.'s upfront SAT decomposition can be large.

More sophisticated was the introduction of *Lazy Clause Generation* (LCG) [83] which improves upon Katsirelos and Bacchus [62] and Tamura et al. [107] by:

- Allowing equality- or bounds-based reasoning as appropriate, by defining a comprehensive Boolean model of the integer domains, combining the above approaches.
- Lazily decomposing the CP problem to SAT during solving, so that if the SAT problem is very large then only the necessary subset of it will be solved, and so that the usual well-known CP propagators can be used to identify domain changes.

In this research we will apply, evaluate and improve LCG solving techniques to tackle scheduling and rostering problems. We now look at LCG in more detail.

2.4 Lazy Clause Generation (LCG)

Lazy Clause Generation (LCG) [83] is tackling a CP problem using a modified DPLL procedure which incorporates Conflict Driven Clause Learning (CDCL), First Unique Implication

Point (1UIP) nogoods, and optionally an autonomous search, as described in Sections 2.2–2.2.4. An LCG solver makes search decisions and propagates like a traditional CP solver, but generates explanations in order to build an *implication graph*, so that conflicts can be analyzed. This results in learnt clauses which reduce future search.

In an LCG solver, propagators explain themselves by globally true redundant constraints which are *clauses* or conjunctions of literals. The clauses consist of logical statements about the domains of variables, based on a Boolean model of domains. Given a variable x with domain $D(x)$, we define literals $[x = v]$, $[x \leq v]$ where $v \in D(x)$, and their negations $[x \neq v]$, $[x > v]$. Then, to claim failure or modify domains, a propagator for a constraint $C(x_1, \ldots, x_n)$ must produce a clause over the literals of $x_1, \ldots, x_n$. The clause must be implied by C and be *false* or unit, i.e. must immediately fail or propagate.

Example 2.6 Consider the constraint $x \neq y$. This decomposes to clauses $[x \neq v] \vee [y \neq v]$ for all $v \in D(x) \cap D(y)$. Since the number of clauses may be large, the LCG solver does not post these clauses a priori. Instead it executes a conventional disequality propagator which looks for x or y becoming fixed, e.g. if x becomes fixed to w it wishes to set $D(y) \leftarrow D(y) \backslash w$. To do this it produces the clause $[x \neq w] \vee [y \neq w]$, which is unit at this point in search, i.e. given $[x \neq w]$ is *false*, the clause can only be satisfied by setting $[y \neq w]$ *true*. In an LCG solver asserting the literal $[y \neq w]$ makes the appropriate domain update $D(y) \leftarrow D(y) \backslash w$. □

Algorithm 2.4 shows an LCG solving algorithm based on the SAT solver of Algorithm 2.1. Compared with the SAT solver the domains and constraints are made more generic, yet the SAT solving can proceed much the same as previously, given the expanded definitions in lines 3–10 which define the meanings of the SAT literals in terms of the CP variables. The main difference is that, in line 16, instead of taking *false* or unit clauses directly from a clause database C, we generate them lazily from the constraints in C.

Another important task, in lines 27 and 33, is to keep the Boolean model of the integer variables consistent with the corresponding integer variable domains. For instance if we set $[x_i = v]$ we have to propagate $[x_i = v] \rightarrow [x_i \neq w], w \neq v$, and similarly if we set $[x_i \leq v]$ we have to propagate $[x_i \leq v] \rightarrow [x_i \leq w], w > v$. Channelling between the equality and bounds representations is via clauses like $[x_i \leq v] \wedge [x_i \neq v] \rightarrow [x_i \leq v - 1]$. Such assignments must be appended to the list G, and thereby added to the current level implication graph, immediately after assigning any literal. They do not have to be stored explicitly in the constraint database C, since they are dynamically generated.

Although the integer variable model propagates at a higher priority than the regular propagators in the system, inconsistencies are possible. For instance, the implied clause $\bigvee_{v \in D_{orig}(x_i)} [x_i = v]$ will usually propagate to set $x_i = v$ when all possibilities other than v have been eliminated from x_i's domain, but if all values are eliminated, it fails with explanation $\bigwedge_{v \in D_{orig}(x_i)} [x_i \neq v] \rightarrow$ *false*. Similarly, bounds inconsistency can cause a

Algorithm 2.4 Example of a Lazy Clause Generation (LCG) solving algorithm using CDCL

inputs:

D_{orig} = map from variable $x_1, \ldots, x_n$ to its initial finite integer domain
$C = \{C_1, \ldots, C_m\}$ where $C_i(x_1, \ldots, x_n)$ is a constraint over $x_1, \ldots, x_n$ with an LCG propagator, and/or a clause over the literals $[x_i = v], [x_i \neq v], [x_i \leq v], [x_i > v]$
A = a set of assumptions as literals to be assigned 1 (*true*) before solving starts

outputs:

satisfiable(S) where S = a map from variable to value satisfying C, or
unsatisfiable(U) where U = a set of assumptions which cannot hold simultaneously

function $LCG(D_{orig}, C, A)$

$D \leftarrow D_{orig}$
define convenience functions and operators
    $V([x = v]) \equiv 1$ if $D(x) = \{v\}$, 0 if $v \notin D(x)$, *unset* otherwise,
    $V([x \neq v]) \equiv 0$ if $D(x) = \{v\}$, 1 if $v \notin D(x)$, *unset* otherwise,
    $V([x \leq v]) \equiv 1$ if $y \leq v \; \forall y \in D(x)$, 0 if $y > v \; \forall y \in D(x)$, *unset* otherwise,
    $V([x > v]) \equiv 0$ if $y \leq v \; \forall y \in D(x)$, 1 if $y > v \; \forall y \in D(x)$, *unset* otherwise,
    $V([x = v]) \leftarrow 1 \equiv D(x) \leftarrow \{v\}$,
    $V([x \neq v]) \leftarrow 1 \equiv D(x) \leftarrow D(x) \backslash \{v\}$,
    $V([x \leq v]) \leftarrow 1 \equiv D(x) \leftarrow \{y \in D(x) : y \leq v\}$,
    $V([x > v]) \leftarrow 1 \equiv D(x) \leftarrow \{y \in D(x) : y > v\}$
$level \leftarrow 0$ # current decision level
$stack \leftarrow []$ # saved assignments per decision level
$G \leftarrow [(assumption, \ell) : \ell \in A]$ # list of assignments for current level
for each $\ell \in A$ **do** $V(\ell) \leftarrow 1$ # assign (assume) literals
while *true* **do**
    if an LCG propagator for a constraint in C returns a *false* or unit clause L **then**
        if L is *false* **then**
            if $level = 0$ **then return** $unsatisfiable(\{\neg \ell : \ell \in analyze\_final(L)\})$
            $L \leftarrow analyze(G, L)$ # new clause from implication graph
            $C \leftarrow C \cup \{L\}$ # save new clause in database
            $level \leftarrow$ smallest decision level such that L would be unit
            $(D, G) \leftarrow stack[level + 1]$ # restore earlier level assignments
            $stack \leftarrow stack[1..level]$ # backtrack (discard assignments)
        let $\{\ell_1, \ldots, \ell_k\} = L$ such that $V(\ell_1) = 0, \ldots, V(\ell_{k-1}) = 0, V(\ell_k) = unset$
        $G \leftarrow G + [(\{\ell_1, \ldots, \ell_{k-1}\}, \ell_k)]$ # add to implication graph
        $V(\ell_k) \leftarrow 1$ # assign (propagate) literal
        additional propagation $\ell_k \rightarrow \cdot$ for integer variable model
    else if programmed or autonomous search chooses an undefined literal ℓ **then**
        $level \leftarrow level + 1$ # branch (new decision level)
        $stack \leftarrow stack + [(D, G)]$ # save current level assignments
        $G \leftarrow [(decision, \ell)]$ # add to implication graph
        $V(\ell) \leftarrow 1$ # assign (decide) literal
        additional propagation $\ell \rightarrow \cdot$ for integer variable model
    else return $satisfiable(\{x \mapsto v : x \mapsto \{v\} \in D\})$

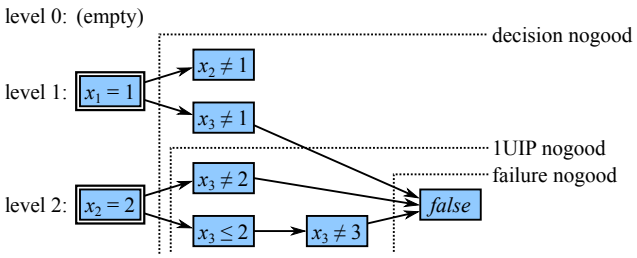


Fig. 2.4 Example implication graph created by Algorithm 2.4, showing several different nogoods

failure like $[x_i \leq v] \wedge [x_i > v + 1] \rightarrow \mathit{false}$. These cases require a jump to the conflict handling routine at lines 18–23, omitted in the pseudocode for simplicity.

LCG allows programmed search as in CP, or autonomous search as in SAT, as shown in line 28 of the algorithm. Since SAT style autonomous search is extremely good, and since it directly improves the effectiveness of learning by focusing the search on recent highly constrained subproblems, the autonomous search is preferred except in particular cases. However, for a few CP problems which have a very efficient programmed search, the programmed search gives an advantage over SAT. We should also use the appropriate type of search in LCG when making direct comparisons with CP or SAT solvers.

2.4.1 Conflict analysis in LCG

We now examine the implication graphs produced by Algorithm 2.4. These are expressed in terms of the Boolean model of domains, allowing LCG to use the same conflict analysis procedure as SAT, shown as Algorithm 2.2 previously. Hence LCG uses 1UIP nogoods [73], which are effective in SAT. We now discuss the choice of nogood in more detail.

We define the *failure nogood* as the direct cause of the conflict before conflict analysis. It is an existing clause of the problem. Although it might be an original problem clause or a previously learnt clause, or be lazily generated e.g. from an original problem constraint or from the integer variable model, we consider it an existing clause, and we would not save it to the clause database as a learnt clause, since this would give no advantage.

Example 2.7 Consider the Constraint Program $\mathit{alldifferent}([x_1, x_2, x_3]) \wedge (x_3 \leq x_2)$ where $x_1, x_2, x_3 \in 1..3$. As the first search decision the solver sets $x_1 = 1$ and propagates that $x_2, x_3 \neq 1$. As the second search decision the solver sets $x_2 = 2$ and propagates that $x_3 \neq 2$ and $x_3 \leq 2$ (we skip some intermediate bounds reasoning for simplicity).

Then the integer variable model fails due to the implicit constraint that x_3 must take some value in $D_{orig}(x_3)$, with explanation $[x_3 \neq 1] \wedge [x_3 \neq 2] \wedge [x_3 \neq 3] \rightarrow \mathit{false}$. This is the failure nogood, i.e. the direct cause of the failure. We show the resulting implication graph Figure 2.4. The cut labelled 'failure nogood' encodes the 3 literals on the LHS of the failure nogood, as the direct predecessors to the cut, or arcs crossing the cut.

The cut labelled '1UIP nogood' encodes an indirect cause of failure $[x_3 \neq 1] \wedge [x_2 = 2] \rightarrow \mathit{false}$. This contains only a single literal from the current decision level, and would propagate as the implication $[x_3 \neq 1] \rightarrow [x_2 \neq 2]$ at the previous choicepoint. □

Any cut in the implication graph which isolates *false* gives a nogood. But, some nogoods are better than others. The failure nogood is the most 'accurate', since it directly encodes the reason for failure. The 1UIP nogood is less accurate, that is to say it is weaker, because it does not encode all of the possible ways the failure nogood could be falsified. We accept this because of its desirable property of cutting off the failed subproblem.

Prior to 1UIP, some solvers used *decision nogoods* [36] which are a minimal list of decisions involved in the current failure, that is, a cut which isolates enough decisions to isolate *false*. For instance, the *PaLM* solver [60] examined in Chapter 4 uses these.

Example 2.8 Referring to Example 2.7, the cut labelled 'decision nogood' encodes the decision nogood $[x_1 = 1] \wedge [x_2 = 2] \rightarrow \mathit{false}$. Here all decisions are involved in the failure. □

The decision nogood is even less accurate than the 1UIP nogood, since in turn it does not encode all of the ways the 1UIP nogood could be falsified. Indeed it is clear in our example that the decision nogood was simply a description of the search which resulted in failure, rather than a description of why the search failed. Also, if the decision nogood includes nearly all decisions, then the situation it describes is unlikely to occur again.

2.5 Typical constraints in LCG

In this section we briefly examine several primitive and several global constraints commonly implemented by LCG solvers, which we will use in later chapters. In particular we look at some typical explanations for each constraint. Since this is only intended as an introduction to LCG concepts, we do not examine all propagation algorithms in detail, and nor do we look at every possible direction of propagation, consistency level, etc.

2.5.1 The *linear* constraint in LCG

The most important constraint in LCG is the *linear* constraint, which we use extensively in Chapters 3 through 5. Indeed most CP models can be linearized [14] and hence solved with only the *linear* constraint. The LCG version of *linear* was introduced by Ohrimenko et al. [83]. It is usually propagated and explained to bounds($\mathcal{R}$) consistency.

The *linear* constraint takes the form $c_1x_1 + \ldots + c_nx_n \geq d$ where $c_1, \ldots, c_n$ and d are integer constants and $x_1, \ldots, x_n$ are integer CP variables. The '$\leq$' form is available by negation, and the '$=$' form is available as the conjunction of '$\leq$' and '$\geq$', a decomposition which does not hinder propagation since the '$\leq$' and '$\geq$' constraints are independent.

An LCG solver checks the *linear* constraint by substituting the most generous possible values for $x_1, \ldots, x_n$. Without loss of generality we consider the '$\geq$' form with positive coefficients. Then letting $x_i \leftarrow \mathrm{ub}(x_i)$, we can evaluate the LHS. If the LHS $> d$ then there is slack in the constraint, if the LHS $< d$ then the constraint is violated.

Propagation proceeds by considering each variable x_i in turn and how it could use up the slack in the constraint. Suppose $m_j = \mathrm{ub}(x_j), j \in 1..n$ are the upper bounds in the present subproblem, taken as constants, and $s = \sum_{j=1}^{n} c_j m_j - d$ is the slack. Then the minimum possible value of x_i is $m_i - \lfloor s/c_i \rfloor$. The clausal explanation for this would be

$$\bigwedge_{j \in 1..n, j \neq i} [x_j \leq m_j] \rightarrow [x_i \geq m_i - \lfloor s/c_i \rfloor]$$

The explanation can be strengthened by *lifting*, that is by artificially increasing some of the $\mathrm{ub}(x_j), j \neq i$. The lifted explanation still propagates provided the increase is small enough to be absorbed by the roundoff in the *floor* operation [83, Example 18].

2.5.2 The *max* constraint in LCG

The *max* constraint, which we use in Section 5.11.4, sets a given variable to the maximum value of some collection of variables, written $y = max([x_1, \ldots, x_n])$. We choose this as a simple example of a primitive constraint with an obvious LCG encoding.

The bounds($\mathcal{Z}$)-consistent version of *max* decomposes to $y \geq max(\ldots)$ and $y \leq max(\ldots)$ without hindering propagation. We consider only the '$\leq$' part, since the '$\geq$' part has an obvious decomposition to *linear* as $y \geq x_1 \wedge \ldots \wedge y \geq x_n$.

The most important direction of propagation for the $y \leq max(\ldots)$ form is setting $\mathrm{ub}(y)$ from $\mathrm{ub}(x_1), \ldots, \mathrm{ub}(x_n)$. Given $m = \max_{i=1}^{n} \mathrm{ub}(x_i)$, clearly y can be no larger than this. The explanation for this propagation is $\bigwedge_{i \in 1..n} [x_i \leq m] \rightarrow [y \leq m]$.

2.5.3 The *element* constraint in LCG

The *element* constraint, which we use in Chapters 4 and 5, looks up some variable from an array by index, which we write $y = a[x]$. We consider the domain-consistent version. We let I be the index set of a and we assume the constraint is total, that is $D(x) \subseteq I$.

The most important direction of propagation is setting $y \neq v$ based on $D(x)$. This occurs when x cannot take any value such that $a[x] = v$. If a is constant, the explanation is $\bigwedge_{i \in I, a[i]=v} [x \neq i] \rightarrow [y \neq v]$. If a consists of integer CP variables, the explanation lists and eliminates all tuples $(i, a[i])$ by stating that either x cannot be i or $a[i]$ cannot be v,

$$\bigwedge_{i \in I} \begin{cases} [x \neq i], & i \notin D(x) \\ [a[i] \neq v], & \text{otherwise} \end{cases} \rightarrow [y \neq v].$$

Similar explanations based on listing and somehow eliminating each possible tuple apply to *element2d*, the 2-dimensional case, which we use in Section 4.10.2.

2.5.4 The *alldifferent* and *gcc* constraints in LCG

The *alldifferent* constraint [111], which we examine in more detail beginning with Section 3.2, is given a collection of variables, each variable must take a different value. We write this $alldifferent([x_1, \ldots, x_n])$. The simplest LCG encoding is, if x_i becomes assigned to v for some $i \in 1..n$, then eliminate v from other domains by $[x_i = v] \rightarrow [x_j \neq v], j \neq i$.

The Global Cardinality Constraint (GCC) [12] is a generalization of *alldifferent* which we write $gcc([x_1, \ldots, x_n], [c_1, \ldots, c_m])$. This states that each value $j \in 1..m$ occurs c_j times, that is $\sum_{i \in 1..n}[x_i = j] = c_j$ taking $[x_i = j]$ as $1 = true$, $0 = false$. We assume for simplicity that $D(x_i) \subseteq 1..m, i = 1..n$, the so-called *closed* version of the constraint.

The most important direction of propagation is based on the upper bounds $\text{ub}(c_j), j \in 1..m$. Let $u = \text{ub}(c_j)$ for the present subproblem, taken as a constant. Suppose u variables with index set U are already assigned to j, that is $x_i = j, i \in U$ where $|U| = u$. Then remaining variables cannot equal j. The explanation is $[c_j \leq u] \wedge \bigwedge_{i \in U}[x_i = j] \rightarrow [x_k \neq j], k \notin U$, clearly a generalization of the *alldifferent* explanation which has $c_j = 1, j \in 1..m$.

2.5.5 Reified constraints in LCG

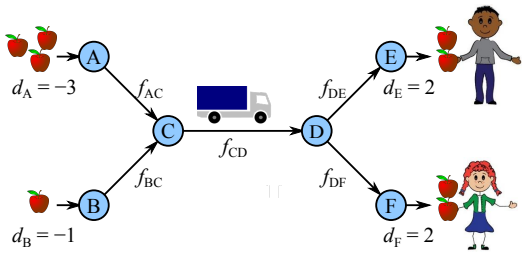
Given a constraint C the *reified* form of the constraint is $a \leftrightarrow C$ for some Boolean CP variable a, that is the constraint holds if and only if a is *true*. This decomposes to $a \rightarrow C \wedge a \leftarrow C$. The *half-reified* form is just the forward direction, $a \rightarrow C$. Feydy et al. show that any propagator P for a constraint C has a corresponding half-reified propagator P' for $a \rightarrow C$ [44].

Half-reification is useful for decomposition, e.g. the constraint $y \leq max([x_1, \ldots, x_n])$ of Section 2.5.2 has a decomposition $a_1 \vee \ldots \vee a_n$ where $a_i \rightarrow (y \leq x_i)$. This decomposition is weaker, but would be essential in an LCG system which does not implement *max* yet. Half-reification is also useful for *assumptions*, as we discuss in Section 5.2.2.

When we derive P' from P in an LCG system, we take the explanations of P' from the explanations of P as follows. Suppose a is *true*, then P' executes P. If P produces an explanation $E \rightarrow f$ where E is a set of literals as preconditions for a propagation or failure f, then the corresponding explanation produced by P' would be $a \wedge E \rightarrow f$.

2.6 Network Flows in CP

Network Flows arise in Combinatorial Optimization problems such as transportation, logistics, mining, mixing, etc. We examine Network Flows in CP in Chapter 4. A good reference on Network Flows is Ahuja et al. [3], we summarize the key concepts here.



(a) Example flow network

$$\begin{bmatrix} -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 \\ 1 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 & -1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f_{AC} \\ f_{BC} \\ f_{CD} \\ f_{DE} \\ f_{DF} \end{bmatrix} = \begin{bmatrix} -3 \\ -1 \\ 0 \\ 0 \\ 2 \\ 2 \end{bmatrix}$$

(b) Equivalent system of equations

Fig. 2.5 Example flow network showing the flow conservation constraints

Flow problems have a directed graph, as a set of nodes and a set of arcs connecting them, in which each arc carries a flow subject to an upper bound and sometimes a lower bound on each arc, and flow is conserved at each node. Flow problems can also have a cost per unit flow in each arc, from which the cost of any given solution can be calculated.

Non-zero flow can be introduced into the network by (i) dedicated source and sink nodes, as in *max flow* problems, (ii) non-zero lower bounds, as in *circulation* problems, or (iii) supplies or demands at each node, as in *minimum cost flow* problems. The form used is a matter of convenience, since transformations exist between them [15].

The directed graph can be written $G = (N, A)$ where N is a set of nodes and A is a set of arcs as pairs of nodes. We introduce a slightly more advanced version of this representation in Section 4.2. The directed graph can also be represented as an incidence matrix $\mathbf{A}$ with 2 non-zeros per column, where $A_{ij} = \pm 1$ indicates arc j goes to/from node i, and optionally a demand vector $\mathbf{d}$. We introduce this representation more formally in Section 4.5.

Example 2.9 Figure 2.5 shows an example Network Flow problem graphically and as a matrix using demands. Nodes A and B represent suppliers who can provide 3 apples and 1 apple respectively. Nodes E and F represent children each of whom require 2 apples. The intermediate nodes and arcs represent a transportation network, so that e.g. f_{CD} is the number of apples travelling from C to D, which must be 4 due to flow conservation.

The matrix representation has exactly two non-zeros per column, with the columns representing the arcs AC, BC, CD, DE, DF and the rows representing the nodes A, B, C, D, E, F in that order. Each row gives a flow conservation equation for a node, e.g. at node C we have $f_{AC} + f_{BC} - f_{CD} = 0$, where the LHS is the flow entering the node less the flow leaving the node, and the RHS is the demand (positive) or supply (negative) for the node. □

Flow problems can be embedded in CP, either (i) by modelling a real-world flow such as a transportation or mixing problem, or (ii) by any arbitrary set of linear equations which encodes a valid incidence matrix, or (iii) by decomposition of global constraints such as *soft_alldifferent*, *gcc* or *sliding_sum* which can be expressed as flow [67, 91, 112].

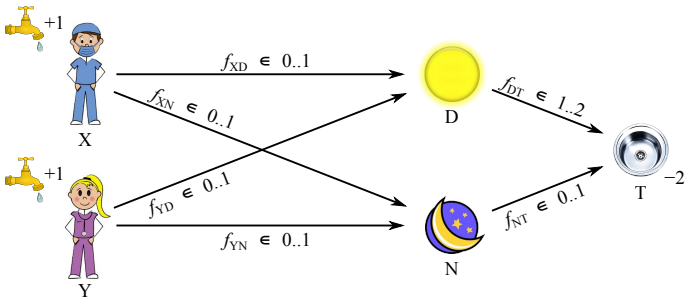


Fig. 2.6 Example flow network encoding a *gcc* constraint

2.6.1 Modelling with Network Flows

The Global Cardinality Constraint (GCC), briefly introduced in Section 2.5.4, encodes assignment problems, e.g. assigning workers to tasks. Given workers $X_1, \ldots, X_n$ and tasks $Y_1, \ldots, Y_m$, let $x_1, \ldots, x_n \in 1..m$ such that $x_i = j$ means worker X_i performs task Y_j. Then $gcc([x_1, \ldots, x_n], [c_1, \ldots, c_n])$ means c_j workers are required for task Y_j.

The *gcc* constraint decomposes to flow as follows. Source nodes $X_1, \ldots, X_n$ represent workers, and supply one unit of flow each, meaning each worker undertakes exactly one task. Transit nodes $Y_1, \ldots, Y_n$ represent tasks, through which one unit of flow passes per worker assigned to the task. Sink node T demands n units of flow. Arcs (X_i, Y_j) where $j \in D(x_i)$ represent assignments, such that one unit of flow means worker i is assigned to task j. Arcs (Y_j, T) represent cardinalities, with flow c_j meaning c_j workers undertake task j.

Example 2.10 Figure 2.6 shows the flow network encoding of a simple example of a GCC constraint, $gcc([x, y], [d, n])$ where $x, y \in 1..2, d \in 1..2, n \in 0..1$.

This represents a nurse rostering problem in which nurses Xavier and Yasmin are assigned shifts x and y respectively (1 = day, 2 = night), d is the number of nurses on day shift (1..2 required), and n is the number of nurses on night shift (0..1 required).

We define nodes X (Xavier) and Y (Yasmin) which source one unit of flow each, denoted '+1'. We define transit nodes D (Day) and N (Night). We define a sink node T which sinks 2 units of flow, denoted '−2'. The flows f_{ij} encode a legal assignment as follows,

$$x = 1 \leftrightarrow f_{XD} = 1 \qquad y = 1 \leftrightarrow f_{YD} = 1 \qquad d = f_{DT}$$
$$x = 2 \leftrightarrow f_{XN} = 1 \qquad y = 2 \leftrightarrow f_{YN} = 1 \qquad n = f_{NT}.$$

The flow balance equations encode the constraints on legal assignments as follows,

(X) $f_{\mathrm{XD}} + f_{\mathrm{XN}} = 1$	Xavier works exactly 1 shift
(Y) $f_{\mathrm{YD}} + f_{\mathrm{YN}} = 1$	Yasmin works exactly 1 shift
(D) $f_{\mathrm{DT}} = f_{\mathrm{XD}} + f_{\mathrm{YD}}$	Between 1 and 2 nurses work day shift
(N) $f_{\mathrm{NT}} = f_{\mathrm{XN}} + f_{\mathrm{YN}}$	Between 0 and 1 nurses work night shift
(T) $f_{\mathrm{DT}} + f_{\mathrm{NT}} = 2$	In total 2 shifts are worked. □

This simplified example illustrates how flow-based constraints can assist in modelling and solving realistic scheduling and rostering problems using CP.

2.6.2 Generic Network Flow constraint

The original generic flow-based global constraint for CP, proposed by Bockmayr et al. [19], is: Given a collection of nodes, arcs, costs, and variables, flow conservation must hold, flow bounds must not be exceeded, and the overall cost of the flow must be assigned to a variable. This is convenient for modelling flow, and allows efficient propagation.

Flow-based constraints were initially implemented by Bockmayr et al. in the earlier constraint system *CHIP* [19]. They can propagate based on feasibility, and based on costs. This means calculating or estimating the minimum and maximum flow in an arc subject to flow conservation and flow bounds (for the feasibility case), or subject to a maximum overall cost of the flow (for the costs case). These authors consider generalized flows, which are useful for mixing problems, we ignore this feature for simplicity.

A more modern implementation of this constraint is by Steiger et al. in *JaCoP* [106]. These authors use a Primal Network Simplex algorithm [3] to solve the Linear Program (LP) associated with the flow, followed by a comprehensive 'arc analysis' procedure which uses Dual Network Simplex on a modified network to estimate minimum and maximum flows. In principle their approach is similar to Bockmayr et al.'s, but more streamlined and presented in more detail. They also provide a partial propagation mode, which uses a heuristic method to select only a subset of arcs for analysis and propagation.

These generic flow propagators did not have explanation capability, so although their usefulness was established in CP, they were not available in an LCG solver. We now look at some possibilities for explaining flow-based constraints.

2.6.3 Explanations for flows without costs

Rochart in his thesis had discussed generating explanations for flow-based constraints without costs [94], and had provided an experimental implementation restricted to *alldifferent* [95] in the solver *PaLM* [60]. *PaLM* is an earlier solver which used some LCG principles, but was not

an LCG solver, since the explanations were generated to support their constraint-retraction search algorithm, rather than as nogoods or for reducing search directly.

To explain flow-based constraints without costs, we extend Rochart's approach to use LCG infrastructure. We examine this in more detail starting at Section 4.2.

2.6.4 Explanations for flows with costs

Linear Programming (LP) solvers routinely provide explanations for conflicts, as a (possibly minimal) set of rows of the LP which cannot hold simultaneously. For instance, *CPLEX* has provided the routine *CPXgetconflict*() since at least version 8.1. If the solving algorithm is Dual Simplex [26], the information about infeasibility comes from a lower-level explanation generated by Dual Simplex, as a weighted sum of the rows of the LP which obviously cannot hold given the bounds of variables appearing in the derived row. For instance, the *CPLEX* routine *CPXdualfarkas*() returns a set of weights like this.

Since Minimum Cost Flow (MCF) [3] is expressible as an LP, we can use the same approach to explain flow-based constraints with costs. We use Dual Network Simplex, a faster specialization of Dual Simplex [26], to obtain certificates as derived *linear* constraints, which we pass to the *linear* propagator in an LCG solver to explain a conflict (or excess cost) by clauses. Note that *CPLEX*'s built-in network solver cannot easily provide such explanations, since it uses Primal Network Simplex [3], and moreover it uses a modified objective for faster convergence, which interferes with certificates.

Davey et al. [31] applied a similar approach to explaining conflicts in combinatorial Binary Integer Programs (BIP). They take the derived row and employ a knapsack algorithm to reduce it to a minimal clause explaining the conflict. Our approach is similar but faster and easier to implement, since LCG already has a *lifting* procedure to produce small clauses from *linear* constraints [83, Example 18]. Also, Davey et al.'s approach does not use clausal propagation or conflict analysis, they generate LP cuts only.

Achterberg [2] also took a similar approach which can be seen as a generalization of Davey et al.'s idea to combinatorial Mixed Integer Programs (MIP). The same comments apply except more so. Whereas LCG defines a comprehensive model for reasoning about changes in the domain of an integer variable, Achterberg partially synthesizes the same reasoning himself, but the last step, of saving the lifted conflict clause to a database and propagating it, is impossible since there is no language to do so. As with Davey et al., it is a cut-generation approach only, but this involves significant compromise due to rounding issues with the integer case. LCG does not require such compromises.

To explain flow-based constraints with costs, we extend Achterberg's approach to use LCG infrastructure, and to read the explanation certificates directly from a Dual Network Simplex data structure. We examine this in more detail starting at Section 4.5.

2.7 Maximum Satisfiability (MaxSAT)

Maximum Satisfiability (MaxSAT) is an optimization version of SAT. MaxSAT problems consist of at least a set of problem variables and a set of soft clauses, with the objective being to find a model (as a valuation to the variables) that minimizes the number of soft clauses falsified by the model – or equivalently, maximizes the number of soft clauses which hold, hence the name MaxSAT, although in this thesis we consider MaxSAT to be a minimization problem (minimizing violation ≡ maximizing satisfaction).

Example 2.11 (MaxSAT) Consider the MaxSAT problem

$$\begin{aligned} &\text{find } x_1, x_2, x_3 \in 0..1 \text{ minimizing } \sum_{i=1}^{6} \begin{cases} 1, & C_i \text{ is violated} \\ 0, & C_i \text{ holds,} \end{cases} \\ &\text{where } C_1 \equiv \neg x_1, \qquad C_2 \equiv \neg x_2, \qquad C_3 \equiv \neg x_3, \\ &\qquad C_4 \equiv x_1 \vee x_2, \qquad C_5 \equiv x_1 \vee x_3, \qquad C_6 \equiv x_2 \vee x_3. \end{aligned} \tag{2.1}$$

The possible models (solutions) to this MaxSAT problem, along with the violation (objective contribution) of each clause, and the resulting objective z, are

x_1	x_2	x_3	C_1	C_2	C_3	C_4	C_5	C_6	z	x_1	x_2	x_3	C_1	C_2	C_3	C_4	C_5	C_6	z
0	0	0	0	0	0	1	1	1	3	0	0	1	0	0	1	1	0	0	2
1	0	0	1	0	0	0	0	1	2	1	0	1	1	0	1	0	0	0	2
0	1	0	0	1	0	0	1	0	2	0	1	1	0	1	1	0	0	0	2
1	1	0	1	1	0	0	0	0	2	1	1	1	1	1	1	0	0	0	3.

(2.2)

This shows that there is no model which satisfies all clauses, or which satisfies all but one clause. Any model which satisfies all but two clauses is an optimal one. □

Partial MaxSAT consists of a set of hard clauses which must hold in all models, plus a set of soft clauses which may be falsified and which constitute the objective. Weighted (partial) MaxSAT has additionally a weight (possibly infinite) associated with each soft clause, in which the objective is to minimize the sum of weights of falsified soft clauses.

Example 2.12 (Weighted partial MaxSAT) Consider Example 2.11 with a different objective,

$$\text{minimize } \sum_{i=1}^{6} \begin{cases} w_i, & C_i \text{ is violated} \\ 0, & C_i \text{ holds,} \end{cases} \quad \text{where } w_1 = 1, w_2 = 2, w_3 = 3, w_4 = \infty, w_5 = \infty, w_6 = \infty.$$

Then considering C_4, C_5, C_6 as hard constraints the possible models are

x_1	x_2	x_3	C_1	C_2	C_3	C_4	C_5	C_6	z
1	1	0	1	2	0	0	0	0	3 (optimal)
1	0	1	1	0	3	0	0	0	4
0	1	1	0	2	3	0	0	0	5
1	1	1	1	2	3	0	0	0	6.

□

In the remainder of this thesis we refer to the *objective* of the problem, denoted z in the examples above, as the numeric violation cost of a given model. This is the count (unweighted MaxSAT), or sum of weights (weighted MaxSAT), of falsified clauses.

2.7.1 Unsatisfiable-core guided optimization

Unsatisfiability-based solving has been an area of active research in the SAT community, leading to MaxSAT solvers which are increasingly effective on suitable problems. In unsatisfiability-based solving, soft constraints are aggressively assumed to hold, with these assumptions selectively relaxed only after proving that certain subsets of the soft constraints cannot hold simultaneously. Such a subset is called an unsatisfiable core.

The unsatisfiability-based MaxSAT solvers use a learning SAT solver backend, because learning solvers can easily diagnose infeasibilities and return unsatisfiable cores, that is, sets of soft constraints that cannot hold simultaneously. Learning solvers prove infeasibility by deriving the empty clause as a resolution tree based on some subset of the problem clauses. Such a set of problem clauses constitutes an unsatisfiable core.

The basic `LSSU` (Linear Search Sat/Unsat) algorithm for CP or MaxSAT solving consists of generating a series of satisfiable problems which yield better and better solutions to the original problem, until no further improvement is possible. By contrast the basic `LSUS` (Linear Search Unsat/Sat) consists of generating a series of unsatisfiable problems until a satisfiable one is generated, yielding a solution which is always optimal.

Fu and Malik proposed the `MSU1` algorithm [48] for unweighted MaxSAT, as a variant of `LSUS` in each problem is generated based on the unsatisfiable core found when solving the previous problem. This has been generalized to `WPM1` [6] and `WMSU1` [69] to tackle weighted MaxSAT, and generalized in other directions, with more intelligent handling of the unsatisfiable cores, to create the more advanced algorithms `MaxRes` [79] and `OLL` [5, 77], both of which won some categories of the MaxSAT 2014 competition [8].

Marques-Silva and Planes also proposed `MSU3` [71] as an `LSUS` variant, and later they also proposed `MSU4` [72] as an `LSSU` variant. These algorithms take the approach of interleaving unsatisfiable core solving with the original `LSUS`/`LSSU` algorithm, which performs the basic optimization after a sufficient set of unsatisfiable cores has been discovered. In these

algorithms the only significant difference from the original LSUS/LSSU algorithm is in the initial assumptions, which are then removed if too strict.

In the following we give some more technical description and categorization of the original and the unsatisfiable core-guided MaxSAT solving algorithms.

2.7.2 Survey of MaxSAT solving algorithms

MaxSAT algorithms keep lower and/or upper bounds on the optimal objective, which are refined by a series of calls to a SAT oracle, passing it a slightly different problem each time based on the current knowledge of the objective. Finding an optimal model for the problem (or for partial MaxSAT, proving that no model exists satisfying the hard clauses of the problem) can be regarded as a byproduct of this refinement process.

In this thesis we consider linear-search MaxSAT algorithms, according to the classification of Morgado et al. [76]. These algorithms maintain current lower and/or upper bounds, and incrementally increase/decrease them as appropriate with each SAT call. By contrast, binary-search MaxSAT algorithms maintain both bounds and bisect the range with each SAT call, which updates either bound depending on satisfiability/unsatisfiability.

2.7.3 Linear-search optimization MaxSAT algorithms

Linear Search Sat/Unsat [76] (LSSU) refers to the basic branch-and-bound approach, e.g. as implemented in *clasp* [51], in which all SAT calls return satisfiable except the last. Each satisfiable call returns a model, which satisfies all hard clauses and may falsify some or all soft clauses. The objective of the most recent model is saved in z_{ub}. Each time z_{ub} decreases, the CNF representation of a new constraint $objective < z_{ub}$ is added to the problem.

Linear Search Unsat/Sat [76] (LSUS) is the opposite scheme to LSSU, in which all SAT calls return unsatisfiable except the last. Initially z_{lb} is assigned 0, and each SAT call solves under the *assumption* that $objective \leq$ the current value of z_{lb}. Then z_{lb} increases by the minimum possible increment with each unsatisfiable SAT call, usually 1 unless a weighted unsatisfiable-core algorithm is in use, hence the first model found is optimal.

2.7.4 Linear-search core-guided unweighted MaxSAT

The algorithms we consider in Chapter 5 are *core-guided* variations on the basic LSSU and LSUS algorithms. The SAT solver is instructed to solve under the *assumptions* that a given set of the soft clauses of the problem hold. Each SAT call returns either (i) a model, which satisfies the hard clauses of the problem and the assumptions, if the problem is satisfiable under the given assumptions; or (ii) an unsatisfiable core, as a set of assumptions that cannot hold simultaneously, if the problem is unsatisfiable under the given assumptions.

We briefly summarize the linear-search core-guided algorithms. Each algorithm is essentially a different scheme for relaxing the assumptions between SAT calls.

In the original core-guided LSUS algorithm MSU1 [48], and more sophisticated variants MaxRes [79] and OLL [77], an assumptions set A consists of initially all soft clauses. If an unsatisfiable core $U \subseteq A$ is found consisting of n assumptions, then A is updated to $A \backslash U$, and then fresh variables, clauses and assumptions are added which require that $n-1$ of the previous assumptions in U must hold. In this way exactly one assumption is relaxed per SAT call. Unlike the basic LSUS algorithm (above), in MSU1/MaxRes/OLL z_{lb} isn't required, so in these algorithms the only assumptions are those concerning soft clauses.

The core-guided LSUS algorithm MSU3 [71] closely resembles the basic LSUS. An assumptions set A initially consists of all soft clauses, and a lower bound z_{lb} is initially set to 0. Each SAT call is constrained by the assumptions A, plus the added assumption $\mathit{objective} \leq z_{lb}$. With each unsatisfiable SAT call returning an unsatisfiable core U, the assumptions set A and lower bound z_{lb} are updated by $A \leftarrow A \backslash U$, $z_{lb} \leftarrow z_{lb} + 1$.

The core-guided LSSU algorithm MSU4 [72] closely resembles the basic LSSU. An assumptions set A initially consists of all soft clauses (or literals controlling those clauses), and an upper bound z_{ub} is initially set to ∞. Each SAT call is constrained by the assumptions A. With each unsatisfiable SAT call returning an unsatisfiable core U, assumptions are updated by $A \leftarrow A \backslash U$, whereas with each satisfiable SAT call returning a model, z_{ub} is updated and a permanent constraint $\mathit{objective} < z_{ub}$ is added to the problem.

2.8 Satisfiability Modulo Theories (SMT)

SAT Modulo Theories (SMT) is an approach to combinatorial problem solving which combines SAT and formal logic. SMT is related to LCG, it is good at similar problems, and tackles them in a similar way. We now explain some differences. A good introduction to SMT is by de Moura and Bjørner [34], we summarize the key concepts here.

SMT problems are expressed in propositional logic over statements in some formal theory T. This gives a SAT problem where the literals of the problem are overloaded with a meaning in T. Then a solution to the SMT problem is a T-consistent solution to the SAT problem. We can solve this using SAT followed by a theory checker for T.

Example 2.13 Consider SMT with a theory of Linear Integer Arithmetic (LIA) [34]. This gives a SAT problem with literals meaning $c_1 x_1 + \ldots + c_n x_n = d$ or $c_1 x_1 + \ldots + c_n x_n \geq d$, with $c_1, \ldots, c_n, d$ constant and $x_1, \ldots, x_n$ taken from some arbitrary set of integer variables.

Now consider the SMT problem $\ell_1 \vee \ell_2$ where $\ell_1 \equiv x = 5$ and $\ell_2 \equiv x \geq 3$:

ℓ_1	ℓ_2	SAT interpretation	LIA interpretation
false	*false*	*false*	$x \neq 5 \wedge x < 3 \equiv T$-consistent
true	*false*	*true*	$x = 5 \wedge x < 3 \equiv T$-inconsistent
false	*true*	*true*	$x \neq 5 \wedge x \geq 3 \equiv T$-consistent
true	*true*	*true*	$x = 5 \wedge x \geq 3 \equiv T$-consistent.

A row is a model if the SAT interpretation is *true* and the T-interpretation is consistent. □

Modern SMT solvers use the $DPLL(T)$ or so-called *lazy approach* [82] to solving, which is very similar to LCG. This requires a SAT solver plus a theory solver for T. Theory solvers are pluggable and different theories are offered for different problems.

In the $DPLL(T)$ approach, a SAT solver tackles the propositional part of the problem, and informs the theory solver when it makes a new assignment. The theory solver incrementally checks the current assignment for consistency, and if inconsistent it provides an explanation, as a set of theory literals which cannot hold simultaneously. The theory solver can also propagate theory literals with explanations and inform the SAT solver as such.

Since a theory solver can fail a subproblem with explanation or propagate a theory literal with explanation, it is exactly like an LCG propagator. LCG is like a special case of SMT with only one theory, a theory of Finite Domain (FD) integer variables. Each propagator in LCG is like a theory solver for some specialized sub-theory of FD variables. On the other hand, SMT understands a broader range of useful but independent theories.

A key difference between LCG and SMT is in the information overloaded onto SAT literals. In LCG this takes a very prescribed format, it can be either $[x = v]$ or $[x \leq v]$ where x is an integer variable and v is an integer constant. In SMT it depends on the theory, and for the LIA theory which is the most similar to LCG, it can be a linear equation of arbitrary length rather than a simple statement about a single variable.

In LCG, all propagators understand each other's literals, so they cooperate to express fine grained statements about variable domains, which combine through conflict analysis and learning. In SMT a literal is a statement about a constraint rather than about a variable, so the explanations are not so fine grained. This can affect the quality of learnt clauses. LCG can also handle simple domain changes without any propagator intervention.

SMT is useful on problems that require a particular theory. For instance, a theory of bitvectors is useful for 'C' program verification [27], which is not well handled by LCG. On the other hand, LCG is likely to be more efficient than SMT for the specific case of CP problems, using FD integer variables, and global constraints known to LCG. Since SMT supports unsatisfiable cores [18], we check this hypothesis in Chapter 5.

Chapter 3

Explaining *alldifferent* Constraints

3.1 Introduction

The *alldifferent* constraint, given a collection of variables, requires that each variable in the constraint takes a different value. The *alldifferent* constraint often occurs in industrial problems like scheduling, rostering, timetabling and planning, for instance the *alldifferent* constraint can assign operators to machines, classes to timeslots, and so forth.

Different propagation algorithms for *alldifferent* are available, corresponding to different tradeoffs between the complexity or execution time of the propagator, versus the strength of the domain filtering and thereby the expected reduction in search. Van Hoeve has a useful discussion of the different algorithms and consistency levels [111].

The most important *alldifferent* propagation algorithms are Régin's original algorithm [90] which propagates to domain consistency (see also Costa [29]), and Lopez-Ortiz et al.'s algorithm [65] which propagates to bounds($\mathcal{Z}$) consistency (see also Puget [88]).

LCG solvers did not have a dedicated *alldifferent* constraint (except through decomposition) until the work described in this chapter, a preliminary version[1] of which appeared in ACSC '12. This is because the usual *alldifferent* propagators for traditional CP cannot be used in an LCG solver without being extended to *explain* themselves by clauses.

In this chapter we remedy the previous deficiency of LCG solvers by adding several dedicated *alldifferent* propagators. We extend the *alldifferent* algorithms that are used in traditional CP [65, 90], to explain themselves by clauses. We give a comprehensive suite of

[1]Downing, N., Feydy, T., Stuckey, P.J.: Explaining alldifferent. In: *Proceedings of the Thirty-fifth Australasian Computer Science Conference - Volume 122*, *ACSC '12*, pp. 115–124. Australian Computer Society, Inc., Darlinghurst, Australia, Australia (2012)

experiments on a set of challenging CP problems, including comparisons with *alldifferent* by decomposition, which has been used in LCG solvers until now.

We show that the tradeoffs for propagation are different between LCG and traditional CP solvers, in terms of the time spent propagating (filtering) and the strength of the resulting filtering, versus the expected reduction in search. We also show that the tradeoffs for decomposition are different between LCG and traditional CP solvers, in terms of globality lost through decomposition and overhead imposed by the decomposition.

3.2 The *alldifferent* constraint

The *alldifferent* constraint [111] takes a set of finite-domain integer problem variables, and requires that each variable takes a different value. The constraint may be defined as

$$\mathit{alldifferent}([x_1, \ldots, x_n]) \equiv \bigwedge_{1 \leq i < j \leq n} x_i \neq x_j. \tag{3.1}$$

An alternative definition uses the domain D of the constraint to check each value,

$$\mathit{alldifferent}([x_1, \ldots, x_n]) \equiv \bigwedge_{i \in D} \sum_{j \in 1..n} (x_j = i) \leq 1 \quad \text{where } D = \bigcup_{i=1}^{n} D(x_i), \tag{3.2}$$

with $(x_j = i)$ the truth value, 0 = *false* or 1 = *true*. Where *alldifferent* is a permutation, that is $n = |D|$, the condition '$\leq$' in the above definition can be strengthened to '='.

This constraint is widely used in Constraint Satisfaction Problems (CSPs), in particular assignment, timetabling and resource-constrained problems, where typically the $x_1, \ldots, x_n$ variables associate tasks with times or resources, and the *alldifferent* constraint states that activities must occur at different times, must use different resources, etc.

Propagating *alldifferent* means given a partial solution to a CSP as a domain for each variable in the CSP, examine whether the domains of variables in an *alldiferent* constraint are consistent with the constraint, and potentially remove unsupported values from domains. We now examine the different propagation algorithms for *alldifferent*.

3.3 Arc-consistent *alldifferent*

The arc-consistent *alldifferent* propagator takes a simplistic view based on Equation (3.1). It simply propagates each disequality $x_i \neq x_j$ separately. Propagating $x_i \neq x_j$ means if x_i becomes fixed to v, that is $D(x_i) = \{v\}$, then set $D(x_j) = D(x_j) \backslash v$ and vice versa. Note that disequalities do not propagate until one of the variables becomes fixed.

To implement the arc-consistent *alldifferent* propagator in an LCG system, we can post $n(n-1)/2$ disequality propagators with explanation capability, or a monolithic *alldifferent*

propagator with explanation capability. We consider these two cases equivalent, as they have equivalent propagation strength and generate equivalent nogoods.

The explaining arc-consistent propagator wakes up when a variable x_i becomes fixed to v, that is $D(x_i) = \{v\}$ for some $i \in 1..n$, and propagates the following set of clauses,

$$[x_i = v] \rightarrow [x_j \neq v] \qquad \forall j \in 1..n, j \neq i, v \in D(x_j). \tag{3.3}$$

These clauses are guaranteed to be unit or *false*, that is to propagate or fail immediately, since $[x_i = v]$ is *true* and $[x_j \neq v]$ is undefined or *false*. Recall that in an LCG solver, setting a literal $[x_j \neq v]$ automatically updates $D(x_j)$ to $D(x_j)\backslash v$ as we require.

If $D(x_i) = D(x_j) = \{v\}$, the above clause is *false* and the solver immediately fails the current subproblem, executes a conflict analysis based on the equivalent conflict clause $[x_i = v] \wedge [x_j = v] \rightarrow$ *false* in order to derive a 1UIP nogood from the conflict clause, backtracks to an appropriate level, enters the nogood, and then continues.

This simplistic arc-consistent *alldifferent* propagator has some advantages compared with the more sophisticated propagators described in the next sections. The propagation overhead is low, since it only cares about fixed variables. Also, the explanations and consequent nogoods are short. Short nogoods are useful because they propagate often.

3.4 Bounds-consistent *alldifferent*

By considering the *alldifferent* constraint as a whole rather than as individual disequalities, stronger propagation, i.e. further domain removals, may be achieved. Here we consider Lopez-Ortiz et al.'s bounds($\mathcal{Z}$)-consistent *alldifferent* propagator [65], and define appropriate explanations allowing it to be extended for use in an LCG solver.

3.4.1 Bounds-consistent *alldifferent* – Hall intervals

The bounds-consistent propagator uses *interval domains* and *Hall intervals* [57]. We take the domain $D(x_i), i \in 1..n$ of a variable in the constraint as $\mathrm{lb}(x_i)..\,\mathrm{ub}(x_i)$, ignoring any holes, i.e. we assume x_i can take any value in $\mathrm{lb}(x_i)..\,\mathrm{ub}(x_i)$. We take $L..U$ as the domain of the constraint where $L = \min_{i=1}^{n} \mathrm{lb}(x_i)$, $U = \max_{i=1}^{n} \mathrm{ub}(x_i)$, so that all $D(x_i) \subseteq L..U$.

Definition 3.1 A Hall interval of a constraint *alldifferent*($[x_1, \ldots, x_n]$) with domain $L..U$, is a pair $(H, a..b)$ with H an index set containing k variables, and $a..b$ a subinterval of $L..U$ of size k, covering the domains of all variables in H. That is, a Hall interval is

$$(H, a..b) \text{ such that } H \subseteq 1..n, a..b \subseteq L..U, |H| = b - a + 1, \bigwedge_{i \in H} D(x_i) \subseteq a..b.$$

Propagation based on Hall intervals relies on all domain values in the subdomain $a..b$ being occupied by a variable whose index is in H. So these values are unavailable for other

Algorithm 3.1 Algorithm to propagate lower bounds of an *alldifferent* constraint

inputs:
$x_1, \ldots, x_n$ = variables in the *alldifferent* constraint
$\mathrm{lb}(x_1)..\,\mathrm{ub}(x_1), \ldots, \mathrm{lb}(x_n)..\,\mathrm{ub}(x_n)$ = current interval domains of the variables

outputs:
success with $\mathrm{lb}(x_1), \ldots, \mathrm{lb}(x_n)$ updated if domains can be made consistent, or
failure(C) where C is a clause over the bounds literals of $x_1, \ldots, x_n$ explaining the failure

function *propagate_alldifferent_bounds*($[x_1, \ldots, x_n]$)
$B \leftarrow \{\mathrm{lb}(x_i) : i \in 1..n\} \cup \{\mathrm{ub}(x_i) + 1 : i \in 1..n\}$ # all interesting subinterval boundaries
$t \leftarrow \{a \mapsto [] : a \in B\}$ # list of variables assigned to bucket (full buckets will merge to right)
$h \leftarrow \{a \mapsto \mathit{false} : a \in B\}$ # whether bucket is Hall interval (*true* buckets will merge to right)
for $i \in 1..n$ ordered by increasing $\mathrm{ub}(x_i)$ **do**
  $k \leftarrow \max\{a : (a \mapsto \cdot) \in t, a \le \mathrm{lb}(x_i)\}$ # k is largest key in t which is $\le \mathrm{lb}(x_i)$
  $v \leftarrow k + |t(k)|$ # v is value which will be assigned to x_i
  if $v > \mathrm{ub}(x_i)$ **then** # v is too large, so bounds not consistent
    $(a, b, C) \leftarrow \mathit{explain\_bounds}(\mathrm{lb}(x_i), \mathrm{ub}(x_i), k, t(k))$ # find Hall start a, description C
    return $\mathit{failure}(C \wedge [x_i \ge a] \wedge [x_i \le b] \rightarrow \mathit{false})$ # abort with clausal explanation
  $t \leftarrow t \backslash \{k \mapsto \cdot\} \cup \{k \mapsto t(k) + [x_i]\}$ # append variable x_i to bucket k
  $\mathit{next\_k} \leftarrow \min\{a : (a \mapsto \cdot) \in t, a > \mathrm{lb}(x_i)\}$ # $\mathit{next\_k}$ is key just after k in t
  if $\mathit{next\_k}$ exists and $= v + 1$ **then** # immediately follows k, so bucket k is full
    $t \leftarrow t \backslash \{k \mapsto \cdot, \mathit{next\_k} \mapsto \cdot\} \cup \{k \mapsto t(k) + t(\mathit{next\_k})\}$ # merge buckets k and $\mathit{next\_k}$
    $v \leftarrow k + |t(k)| - 1$ # v is new last value in bucket k
  $\ell \leftarrow \max\{a : (a \mapsto \cdot) \in h, a \le \mathrm{lb}(x_i)\}$ # ℓ is largest key in h which is $\le \mathrm{lb}(x_i)$
  if $h(\ell)$ **then** # $\mathrm{lb}(x_i)$ falls into a known Hall interval
    $\mathit{next\_}\ell \leftarrow \min\{a : (a \mapsto \cdot) \in h, a > \mathrm{lb}(x_i)\}$ # $\mathit{next\_}\ell$ is key just after ℓ in h
    $(a, b, C) \leftarrow \mathit{explain\_bounds}(\mathrm{lb}(x), \mathit{next\_}\ell - 1, k, t(k))$ # find Hall start a, description C
    propagate $C \wedge [x_i \ge a] \rightarrow [x_i > b]$ # set $x_i \ge \mathit{next\_}\ell$ with explanation
  if $v = \mathrm{ub}(x_i)$ **then** # v is tight, so interval $k..v$ is Hall
    $h \leftarrow h \backslash \{a \mapsto \cdot : (a \mapsto \cdot) \in h, a \in k..v\} \cup \{k \mapsto \mathit{true}\}$ # mark buckets in $k..v$ and merge
return *success*

variables, and can be removed from other domains. In particular, if $\mathrm{lb}(x_i) \in a..b$ then $\mathrm{lb}(x_i)$ increases to $b + 1$, and if $\mathrm{ub}(x_i) \in a..b$ then $\mathrm{ub}(x_i)$ decreases to $a - 1$, for $i \notin H$.

3.4.2 Bounds-consistent *alldifferent* – algorithm

Lopez-Ortiz et al.'s *alldifferent* propagator is based on finding a support and a set of Hall intervals for the constraint based on the interval domains of $x_1, \ldots, x_n$, as follows.

1. Allocate each variable a value in order most-to-least constrained, taking $\mathrm{ub}(x_i)$ as the constrainedness, that is for each $i \in 1..n$ ordered by increasing $\mathrm{ub}(x_i)$, let $x_i = v$ where v is the first available value $v \ge \mathrm{lb}(x_i)$. This finds a support if one exists.
2. Construct the Hall intervals by observing whether $v = \mathrm{ub}(x_i)$, in which case the last available value in an interval has been taken and a Hall interval discovered.
3. Filter by observing whether $\mathrm{lb}(x_i)$ falls into a previously discovered Hall interval, in which case $\mathrm{lb}(x_i)$ can be increased to just past the end of the Hall interval.
4. To prune $\mathrm{ub}(x_1), \ldots, \mathrm{ub}(x_n)$, repeat the above with bounds negated and swapped.

We show a simplified version of Lopez-Ortiz et al.'s algorithm as Algorithm 3.1, the low-level details are in the original paper [65]. We instrumented Algorithm 3.1 to produce explanations, via calls to a utility function *explain_bounds*(), which can be ignored if we just want a high level understanding of Lopez-Ortiz et al.'s original algorithm.

In our version, the interval domain $L..U$ is divided into subintervals, which we call *buckets*. The maps t and h are keyed by the bucket identifier which is the first value in the bucket's subinterval, the last value being the next key $-$ 1. In the original algorithm, t and h are union-find data structures [109] to facilitate fast merging of buckets, we ignore this feature for simplicity. Lopez-Ortiz et al.'s data structure can be recovered from ours and vice versa, except that we record the variables assigned to each bucket, which we use for explanations, whereas in their version, they only keep a remaining-capacity counter.

We use the same example as in Lopez-Ortiz et al. [65], so that the reader can follow along if desired and see what happens to their data structures alongside ours.

Example 3.1 Consider the constraint $\mathit{alldifferent}([x_1, \ldots, x_6])$ with $x_1 \in 3..4, x_2 \in 2..4, x_3 \in 3..4, x_4 \in 2..5, x_5 \in 3..6, x_6 \in 1..6$. The initial buckets are as follows,

$$t = \{1 \mapsto [], 2 \mapsto [], 3 \mapsto [], 5 \mapsto [], 6 \mapsto [], 7 \mapsto []\},$$
$$h = \{1 \mapsto \mathit{false}, 2 \mapsto \mathit{false}, 3 \mapsto \mathit{false}, 5 \mapsto \mathit{false}, 6 \mapsto \mathit{false}, 7 \mapsto \mathit{false}\}.$$

The support is constructed by: Let $x_1 = 3$ putting x_1 in bucket 3 of t,

$$t = \{1 \mapsto [], 2 \mapsto [], 3 \mapsto [x_1], 5 \mapsto [], 6 \mapsto [], 7 \mapsto []\}.$$

Let $x_2 = 2$ putting x_2 in bucket 2 of t, which fills and merges with bucket 3,

$$t = \{1 \mapsto [], 2 \mapsto [x_2, x_1], 5 \mapsto [], 6 \mapsto [], 7 \mapsto []\}.$$

Let $x_3 = 4$ putting x_3 in bucket 2 of t, which fills and merges with bucket 5,

$$t = \{1 \mapsto [], 2 \mapsto [x_2, x_1, x_3], 6 \mapsto [], 7 \mapsto []\}.$$

Last value in bucket 2 of t is $4 = \mathrm{ub}(x_3)$, so mark bucket 2 of h up to value 4 as Hall,

$$h = \{1 \mapsto \mathit{false}, 2 \mapsto \mathit{true}, 5 \mapsto \mathit{false}, 6 \mapsto \mathit{false}, 7 \mapsto \mathit{false}\}.$$

Let $x_4 = 5$ putting x_4 in bucket 2 of t, which fills and merges with bucket 6,

$$t = \{1 \mapsto [], 2 \mapsto [x_2, x_1, x_3, x_4], 7 \mapsto []\}.$$

Bucket 2 of h which contains $\text{lb}(x_4) = 2$ is Hall, so increase $\text{lb}(x_4)$ to next key in $h = 5$. Last value in bucket 2 of t is $5 = \text{ub}(x_4)$, so mark bucket 2 of h up to value 5 as Hall,

$$h = \{1 \mapsto \mathit{false}, 2 \mapsto \mathit{true}, 6 \mapsto \mathit{false}, 7 \mapsto \mathit{false}\}.$$

Let $x_5 = 6$ putting x_5 in bucket 2 of t, which fills and merges with bucket 7,

$$t = \{1 \mapsto [], 2 \mapsto [x_2, x_1, x_3, x_4, x_5]\}.$$

Bucket 2 of h which contains $\text{lb}(x_5) = 3$ is Hall, so increase $\text{lb}(x_5)$ to next key in $h = 6$. Last value in bucket 2 of t is $6 = \text{ub}(x_5)$, so mark bucket 2 of h up to value 6 as Hall,

$$h = \{1 \mapsto \mathit{false}, 2 \mapsto \mathit{true}, 7 \mapsto \mathit{false}\}.$$

Let $x_6 = 1$ putting x_6 in bucket 1 of t, which fills and merges with bucket 2,

$$t = \{1 \mapsto [x_6, x_2, x_1, x_3, x_4, x_5]\}.$$

Last value in bucket 1 of t is $6 = \text{ub}(x_6)$, so mark bucket 1 of h up to value 6 as Hall,

$$h = \{1 \mapsto \mathit{true}, 7 \mapsto \mathit{false}\}.$$

The propagator changed $D(x_4)$ from 2..5 to 5..5 and $D(x_5)$ from 3..6 to 6..6. □

3.4.3 Bounds-consistent *alldifferent* – explanations

Referring to Algorithm 3.1, the explanation function *explain_bounds*() is called in two places, corresponding to failure and propagation. In the first case, we have to produce an explanation clause which is *false*, i.e. immediately fails and backtracks. In the second case, we have to produce an explanation clause which is unit, i.e. immediately propagates.

Referring to Definition 3.1, the explanation function should find a Hall interval $a..b$ and the set of variables H contained in $a..b$, and then construct a logical expression C describing the Hall interval in terms of bounds literals $[x_i \geq v]$ and their '$<$' or '$\leq$' negations,

$$C \equiv \bigwedge_{i \in H} ([x_i \geq a] \wedge [x_i \leq b])$$

which restates Definition 3.1 in literals. If this holds then the values $a..b$ are unavailable. To make this into a *false* or unit clause, we require one extra variable x_i, $i \notin H$, to generate a statement that $b - a + 2$ variables take values in $a..b$, which cannot happen. That is

$$C \wedge [x_i \geq a] \wedge [x_i \leq b] \rightarrow \mathit{false} \text{ where } i \in 1..n, i \notin H. \tag{3.4}$$

Algorithm 3.2 Algorithm to find the start and description of a Hall interval

inputs:

c = first value that needs to be covered by rebuilt Hall interval
b = end of left-maximal Hall interval
k = start of left-maximal Hall interval, also key of bucket containing Hall interval
x = variables in bucket, $x[1]$ takes value k and so on, $|x| \geq b - k + 1$

outputs:

(a, b, C) where $k \leq a \leq c$, $a..b$ = rebuilt interval, C = logical description of interval

function *explain_bounds*(c, b, k, x)

1 $a \leftarrow b + 1$ # $a..b$ is rebuilt Hall interval, initially $\emptyset$
2 $i \leftarrow 0$ # i is number of variables in explanation
3 $n \leftarrow b - k + 1$ # $x[n]$ is last variable in explanation
4 **while** $a > c$ or $i < b - a + 1$ **do**
5 $a \leftarrow \min(a, \text{lb}(x[n - i]))$ # add variable $x[n - i]$ to explanation
6 $i \leftarrow i + 1$
7 **return** $\left(a, b, \bigwedge_{j=n-i+1}^{n}([x[j] \geq a] \wedge [x[j] \leq b])\right)$

The first *explain_bounds*() in Algorithm 3.1 line 8 generates explanation (3.4) with all literals on the LHS *true*. That is, the clause is falsified. The second *explain_bounds*() at line 18 generates explanation (3.4) with all literals on the LHS *true* except $[x_i \leq b]$. That is, the clause is unit. Then $[x_i \leq b]$ cannot be *true*, so the clause propagates to set $[x_i > b]$.

We show the utility function *explain_bounds*() in Algorithm 3.2. This is a little bit subtle. As input we have a guaranteed Hall interval $k..b$ containing variables $x[1..b - k + 1]$, and a simplistic implementation would just return an explanation computed from this.

However, Lopez-Ortiz et al.'s algorithm is designed to find *left-maximal* Hall intervals, with k as small as possible. They do this to improve efficiency, by merging as many buckets as possible. They show in their paper [65] that when keys $k, next\_k$ are merged in t, the deleted key $next\_k$ is *dominated* by k and can't begin a left-maximal Hall interval.

We want explanations to be as short and hence reuseable as possible, so we don't want left-maximal Hall intervals. On the other hand, since we use backward explanations [45], the propagator runs as an ordinary FD propagator in the forward direction, and the same efficiency considerations apply as in Lopez-Ortiz et al.'s work. We handle this by generating a more minimal explanation if possible from c, b, k, x at explanation time.

To rebuild the Hall interval, we start with an empty set H of variables in the Hall interval, and add variables to H working backwards, simultaneously computing $a..b$ the union of the interval domains of the variables in H. We continue until H covers the necessary interval to explain the current failure or propagation, and has become Hall i.e. the number of variables in H catches up to its domain size. In the algorithm we don't maintain H, we just maintain $i = |H|$. Then the set of variables in H is implicitly $x[n - i + 1..n]$.

Example 3.2 Consider the constraint *alldifferent*($[x_1, x_2, x_3, x_4, x_5]$) with $x_1 \in 1..2, x_2 \in 2..3, x_3 \in 2..4, x_4 \in 3..4, x_5 \in 2..4$. Propagating $x_1, \ldots, x_4$ gives

$$t = \{1 \mapsto [x_1, x_2, x_3, x_4]\}, \qquad h = \{1 \mapsto \mathit{true}, 5 \mapsto \mathit{false}\}.$$

Propagating x_5 fails because it tries to allocate the next value in bucket 1 of t which is 5, but $\text{ub}(x_5) = 4$. The initial Hall interval is $(H, k..b) = ([1, 2, 3, 4], 1..4)$, larger than necessary. Algorithm 3.2 rebuilds the Hall interval from right to left to cover $c = 2$ as follows,

add x_4 :	$H = [4]$	$a..b = 3..4$, doesn't cover c	$\lvert H\rvert < a - b + 1$, not Hall
add x_3 :	$H = [3, 4]$	$a..b = 2..4$, covers c	$\lvert H\rvert < a - b + 1$, not Hall
add x_2 :	$H = [2, 3, 4]$	$a..b = 2..4$, covers c	$\lvert H\rvert = a - b + 1$, Hall.

The rebuilt interval now covers $c..b$ i.e. $D(x_5)$ and is Hall, so it can explain the failure,

$$[x_2 \geq 2] \wedge [x_2 \leq 4] \wedge [x_3 \geq 2] \wedge [x_3 \leq 4] \wedge [x_4 \geq 2] \wedge [x_4 \leq 4] \wedge [x_5 \geq 2] \wedge [x_5 \leq 4] \rightarrow \mathit{false}.$$

□

Example 3.3 Consider the previous example except with $x_5 \in 2..5$. Propagation assigns the support $x_5 = 5$, but notices that bucket 1 of h contains $\text{lb}(x_5) = 2$ and is Hall. The Hall interval is rebuilt as previously, giving exactly the same explanation except it is unit,

$$[x_2 \geq 2] \wedge [x_2 \leq 4] \wedge [x_3 \geq 2] \wedge [x_3 \leq 4] \wedge [x_4 \geq 2] \wedge [x_4 \leq 4] \wedge [x_5 \geq 2] \rightarrow [x_5 > 4].$$ □

3.5 Domain-consistent *alldifferent*

By considering the *alldifferent* constraint as a whole with respect to individual values in its variables' domains, rather than just interval domains, even stronger propagation, i.e. further domain removals, may be achieved. Here we consider Régin's domain-consistent *alldifferent* propagator [90], and the explanations proposed by Katsirelos [61] and Rochart et al. [95], which allow Régin's propagator to be extended for use in an LCG solver.

3.5.1 Domain-consistent *alldifferent* – Hall sets

The domain-consistent *alldifferent* propagator uses *Hall sets* [57]. To define Hall sets we first take $D = \cup_{i=1}^{n} D(x_i)$ as the domain of the constraint, so that all $D(x_i) \subseteq D$. In parts of our discussion we will assume $D = 1..m$ for simplicity, where $m \geq n$.

Definition 3.2 A Hall set of a constraint *alldifferent*($[x_1, \ldots, x_n]$) with domain D, is a pair (H, S) with H an index set containing k variables, and S a subdomain of D of size k, covering

the domains of all variables in H. That is, a Hall set is

$$(H, S) \text{ such that } H \subseteq 1..n, S \subseteq D, |H| = |S|, \bigwedge_{i \in H} D(x_i) \subseteq S.$$

Propagation based on Hall sets relies on all domain values in the subdomain S being occupied by a variable whose index is in H. So these values are unavailable for other variables, and can be removed from other domains. That is, $D(x_i)$ becomes $D(x_i) \backslash S$ for $i \notin H$.

3.5.2 Domain-consistent *alldifferent* – bipartite matching

Régin's propagator uses matching theory in bipartite graphs. The best reference on bipartite graphs is Lovász and Plummer [66], we summarize the concepts here.

The *alldifferent* graph $G = (\mathbf{X}, \mathbf{Y}, \mathbf{A})$ has nodes $\mathbf{X} = \{\mathrm{X}_1, \ldots, \mathrm{X}_n\}$ corresponding to variables $x_1, \ldots, x_n$ in the constraint, nodes $\mathbf{Y} = \{\mathrm{Y}_1, \ldots, \mathrm{Y}_m\}$ corresponding to the domain $D = 1..m$ of the constraint, and undirected arcs $\mathbf{A} = \{(\mathrm{X}_i, \mathrm{Y}_j) \in \mathbf{X} \times \mathbf{Y} : j \in D(x_i)\}$ corresponding to legal matchings of (*variable*, *value*) for the current CP subproblem.

We define a matching as a collection M of undirected arcs in $\mathbf{A}$ in which no node in $\mathbf{X}$ or $\mathbf{Y}$ occurs more than once. Then a matching M defines a partial solution $x_i = j, (\mathrm{X}_i, \mathrm{Y}_j) \in M$ to the *alldifferent* constraint. A *complete matching*, or *maximal matching* in Régin's notation, has cardinality n such that every node in $\mathbf{X}$ is matched with some legal node in $\mathbf{Y}$.

Given a graph G and a matching M which may be incomplete, we can calculate the *residual graph* of the matching. The residual graph G^* is a convenient directed-graph representation of which arcs can legally be added to or removed from M, defined as

$$G^* = \{(\mathrm{Y}_j, \mathrm{X}_i) : (\mathrm{X}_i, \mathrm{Y}_j) \in M\} \cup (\mathbf{A} \backslash M). \tag{3.5}$$

The first part of (3.5) lists the arcs already in M which can be taken out. The second part of (3.5) lists the arcs not already in M which can be added. An undirected arc $(\mathrm{X}_i, \mathrm{Y}_j)$ in M cannot really be removed from M if $D(x_i) = \{j\}$, but we ignore this for simplicity.

3.5.3 Domain-consistent *alldifferent* – checking

Régin uses Hopcroft and Karp's algorithm [58] to find a complete matching and hence a support for *alldifferent*. We use Edmonds and Karp's algorithm [39] specialized to bipartite matchings, which is slower but simpler, and most importantly, produces an infeasibility certificate when no complete matching exists, which we use for explanations.

Algorithm 3.3 shows how we find a complete matching. If we have a previous matching M, we take out any edges $(\mathrm{X}_i, \mathrm{Y}_j) \notin \mathbf{A}$ which are no longer valid for current domains because $j \notin D(x_i)$, or we can start with $M = \emptyset$. Then for each unmatched variable $i \in 1..n$ such that $(\mathrm{X}_i, \cdot) \notin M$, we call *extend_matching*(M, X_i) to generate a match for X_i. When all nodes in $\mathbf{X}$ are matched, M is completed, showing that the constraint is supported.

Algorithm 3.3 Edmonds and Karp procedure to extend an incomplete matching

inputs:

$G = (\mathbf{X}, \mathbf{Y}, \mathbf{A})$ = *alldifferent* graph, where $\mathbf{A}$ encodes domains $D(x_1), \ldots, D(x_n)$
M = incomplete matching as a subset of $\mathbf{A}$, maybe empty (no assignments yet)
G^* = residual graph of M encoding allowable changes to M given domains in $\mathbf{A}$
x = variable node to be assigned, where $x \in \mathbf{X}$, $(x, \cdot) \notin M$

outputs:

success(M) with M extended to assign x, or
failure(Q) with Q the set of nodes searched

function *extend_matching*(M, x)

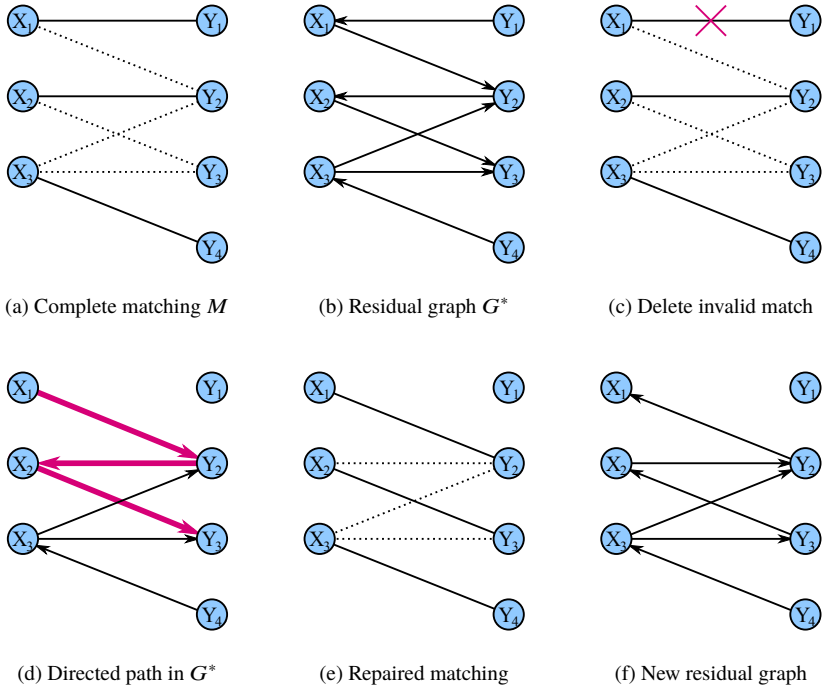
```
1  Q ← [x]                          # Q is queue of nodes to visit in breadth-first order
2  P ← [∅]                          # P[i] is set of directed arcs taken to reach node Q[i]
3  i ← 1                            # Q[1..i − 1] are already visited, Q[i] is next to visit
4  while i ≤ |Q| do
5      forall j such that (Q[i], j) ∈ G* do
6          path = P[i] ∪ {(Q[i], j)}                  # append next arc to current path
7          if j ∈ Y and (·, j) ∉ M then               # success, reached free value node
8              return success(M\{(x, y) : (y, x) ∈ path ∩ Y × X} ∪ (path ∩ X × Y)
9          Q ← Q + [j]
10         P ← P + [path]
11     i ← i + 1
12 return failure(Q)
```

The matching algorithm works by starting at X_i and finding a directed *augmenting path* in the residual graph G^* to any free (unmatched) value node, by breadth-first search. Note that in Régin's notation, an augmenting path in G^* is an *alternating path* in G. Following the augmenting path shows how to move conflicting assignments in M out of the way, in order to eventually free up a value node adjacent to X_i which can be matched to X_i.

Example 3.4 Consider the constraint *alldifferent*$([x_1, x_2, x_3])$ with $x_1 \in \{1, 2\}$, $x_2 \in \{2, 3\}$, $x_3 \in \{2, 3, 4\}$. This has a support $x_1 = 1, x_2 = 2, x_3 = 4$ shown as the complete matching M in Figure 3.1a, where solid arcs are in M, and dotted arcs are in $\mathbf{A}$ but not M. The same information is encoded by the directions of the arcs in the residual graph in Figure 3.1b. Suppose propagation sets $x_1 \neq 1$, then the *alldifferent* propagator wakes up and deletes the invalid arc $(\mathrm{X}_1, \mathrm{Y}_1) \notin \mathbf{A}$ from M, as shown in Figure 3.1c.

To repair M, Algorithm 3.3 looks for a directed path in G^* from X_1 to any free value node, finding the path shown in Figure 3.1d. Following this path it adds $(\mathrm{X}_1, \mathrm{Y}_2)$, removes $(\mathrm{X}_2, \mathrm{Y}_2)$ and adds $(\mathrm{X}_2, \mathrm{Y}_3)$ to the matching. This gives a new support $x_1 = 2, x_2 = 3, x_3 = 4$ for *alldifferent*, shown as the repaired matching in Figure 3.1e. Each augmented arc reverses its direction in the residual graph of the new support, as shown in Figure 3.1f. □

Suppose the contraint has no support, then we have to produce an explanation clause which is *false*, i.e. immediately fails and backtracks. In this case some variable $i \in 1..n$ cannot be



(a) Complete matching M (b) Residual graph G^* (c) Delete invalid match

(d) Directed path in G^* (e) Repaired matching (f) New residual graph

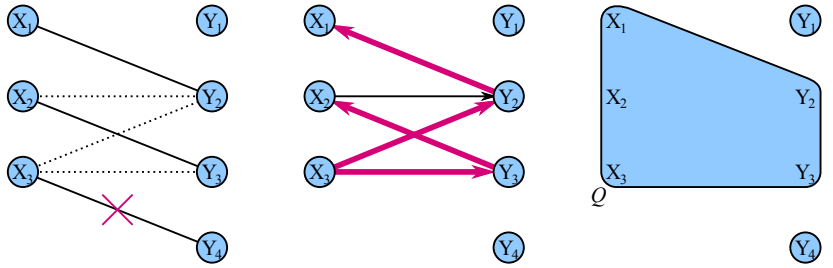
Fig. 3.1 Example of extending a matching using Algorithm 3.3

matched, and Algorithm 3.3 returns *failure*(Q), where Q is an infeasibility certificate, as a list of the nodes searched to establish that no free value node is reachable from X_i. We show that Q encodes a Hall set which can be used to explain the failure.

Theorem 3.1 *If extend_matching*(X_i, M) *returns failure*(Q) *for some* $i \in 1..n$*, then* Q *encodes a Hall set* (H, S) *as per Definition 3.2, by* $Q = \{\mathrm{X}_i\} \cup \{\mathrm{X}_j : j \in H\} \cup \{\mathrm{Y}_k : k \in S\}$*. The Hall set does not include* x_i *in its variables, but covers* $D(x_i)$ *with its domain.*

Proof Let $Q' = Q \backslash \{\mathrm{X}_i\}$. We show that $Q' \cap \mathbf{X}$ and $Q' \cap \mathbf{Y}$ are matched to each other in M. This occurs because every node in $Q' \cap \mathbf{Y}$ is matched by assumption, and its match is reachable from $\mathbf{Y}$ in G^* hence is visited, and a node in $\mathbf{X}$ is only reachable in G^* from its match in $\mathbf{Y}$. Therefore $|Q' \cap \mathbf{X}| = |Q' \cap \mathbf{Y}|$, showing that $|H| = |S|$ in the theorem, the first condition of Definition 3.2. Further, given $j \in 1..n$ such that $\mathrm{X}_j \in Q'$, all nodes $\mathrm{Y}_k, k \in D(x_j)$ are either matched to X_j or reachable from X_j by G^*, and so are in $Q' \cap \mathbf{Y}$. Hence S in the theorem covers the domains of variables in H, the second condition of Definition 3.2. Finally, all value nodes $\mathrm{Y}_k, k \in D(x_i)$ are reachable from X_i, hence S covers $D(x_i)$. □

The explanation clause states that $x_j \in S$ for all $j \in H \cup \{i\}$, where x_i is the variable that failed to be matched. That is, $|S| + 1$ variables take values in S, which cannot happen. We



(a) Delete invalid match (b) Breadth-first search (c) Set Q of nodes searched

Fig. 3.2 Example of failure to extend a matching using Algorithm 3.3

can encode that $x_j \in S$ using equality literals, giving the simplistic explanation

$$\bigwedge_{j \in H \cup \{i\}} \bigvee_{k \in S} [x_j = k] \rightarrow \mathit{false},$$

but since an explanation must be clausal, it is more convenient to encode that $x_j \notin D \backslash S$ using disequality literals, giving Katsirelos's [61] and Rochart et al.'s [95] explanations,

$$\bigwedge_{j \in H \cup \{i\}} \bigwedge_{k \in D \backslash S} [x_j \neq k] \rightarrow \mathit{false}.$$

Example 3.5 Continuing Example 3.4, suppose propagation sets $x_3 \neq 4$, then the *alldifferent* propagator wakes up and deletes the invalid arc $(\mathrm{X}_3, \mathrm{Y}_4) \notin \mathbf{A}$ from M, as shown in Figure 3.2a. To repair M, Algorithm 3.3 looks for a directed path in G^* from X_3 to any free value node, searching the paths shown in Figure 3.1b by breadth-first search before giving up. The infeasibility certificate is $Q = [\mathrm{X}_3, \mathrm{Y}_2, \mathrm{Y}_3, \mathrm{X}_1, \mathrm{X}_2]$ as shown in Figure 3.2c. This encodes the unmatched variable node X_3 plus the Hall set (H, S) where $H = \{1, 2\}, S = \{2, 3\}$. That is, $D(x_1) \cup D(x_2)$ has size 2 and covers $D(x_3)$. The overall domain of the constraint is $D = \{1, 2, 3, 4\}$, so the explanation encodes that $x_1, x_2, x_3 \notin D \backslash S = \{1, 4\}$,

$$[x_1 \neq 1] \wedge [x_1 \neq 4] \wedge [x_2 \neq 1] \wedge [x_2 \neq 4] \wedge [x_3 \neq 1] \wedge [x_3 \neq 4] \rightarrow \mathit{false}. \qquad \square$$

3.5.4 Domain-consistent *alldifferent* – propagating

If the checking phase is successful we have a support for *alldifferent* as a valuation $x_i = j, (\mathrm{X}_i, \mathrm{Y}_j) \in M$. We then wish to check support for individual domain values. This means, given $i \in 1..n, j \in D(x_i)$, is there a support for *alldifferent* in which $x_i = j$?

Equivalently, given $(\mathrm{X}_i, \mathrm{Y}_j) \in \mathbf{A}$, is there any complete matching M' which includes $(\mathrm{X}_i, \mathrm{Y}_j)$? Following Régin we can check this using the residual graph G^* derived from the

current resident matching M by Equation 3.5. An arc $(\mathrm{X}_i, \mathrm{Y}_j) \in \mathbf{A}$ is supported if and only if it can be added to M similarly to Algorithm 3.3, i.e. by following an augmenting path or cycle in G^* to move any conflicting assignments out of the way first.

Given $(\mathrm{X}_i, \mathrm{Y}_j) \in \mathbf{A}$, either (i) it is part of a directed path in G^* from the value node matched to X_i to any free value node, or (ii) it is part of any directed cycle in G^*, or (iii) it is vital, that is, present in all complete matchings including M. If none of these cases hold, then $x_i \neq j$ and we can set $D(x_i) \leftarrow D(x_i) \backslash j$ with an appropriate explanation.

Régin's propagator handles case i with a reachability analysis starting at free nodes. Let $\mathbf{F}$ be the set of nodes from which a free value node is reachable. We find $\mathbf{F}$ by depth-first search of G^* starting at free value nodes and following arcs of G^* in reverse, marking encountered nodes. Then arcs between nodes in $\mathbf{F}$ are those in case i, and are supported.

Régin's propagator handles cases ii-iii by separating the remaining nodes $(\mathbf{X} \cup \mathbf{Y}) \backslash \mathbf{F}$ into Strongly Connected Components (SCCs) using Tarjan's algorithm [108].

Definition 3.3 An SCC of the residual graph G^*, is a maximal subset of the nodes $\mathbf{X} \cup \mathbf{Y}$, in which every node is reachable from every other node in the SCC via directed arcs in G^*.

Corollary 3.1 *Any arc in an SCC of G^* is part of a cycle in G^*, since a cycle containing the arc can always be completed from its head to its tail via remaining arcs in the SCC.*

Then arcs in G^* within SCCs are those in case ii, whereas arcs between SCCs, or from $\mathbf{F}$ to an SCC, represent either vital domain values i.e. case iii, or unsupported domain values i.e. propagations we need to do with an explanation. Note that arcs from an SCC to $\mathbf{F}$ cannot occur, since the SCC would have become part of $\mathbf{F}$ by reverse reachability.

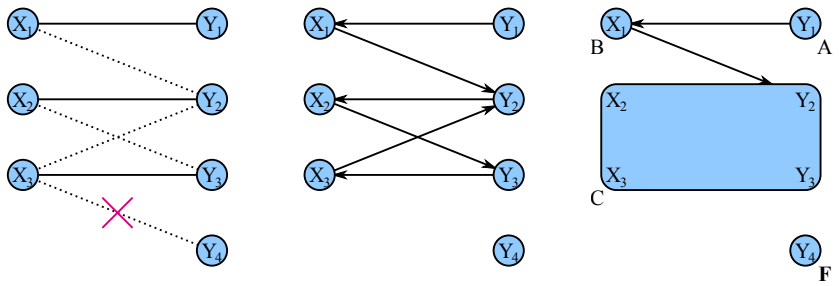
Tarjan's algorithm returns a partition assigning every node, except those in $\mathbf{F}$, to an SCC, with trivial (singleton) SCCs permitted. We instrument the algorithm to also return a Directed Acyclic Graph (DAG) based on a depth-first search order, showing the connectivity of SCCs. The DAG consists of SCCs connected by arcs not in SCCs.

The DAG allows us to produce explanations for the SCC-based propagation. We show that, as Rochart et al. observed [95], we must propagate in the order specified by the DAG, so that the SCCs encode Hall sets to use to explain the unsupported domain values.

Lemma 3.1 *Given $(\mathrm{X}_i, \mathrm{Y}_j) \in M$, i.e. a variable node and its matched value node, either X_i and Y_j occur together in the same SCC, or X_i and Y_j occur separately as trivial SCCs.*

Proof In G^*, node X_i is only reachable from its match Y_j and no other node is reachable from Y_j. If X_i or Y_j are in a cycle, the cycle contains $(\mathrm{Y}_j, \mathrm{X}_i)$ and hence X_i and Y_j. Otherwise, nothing is reachable from Y_j and nothing can reach X_j, so they must be trivial SCCs.

Theorem 3.2 *Any non-trivial SCC A encodes a Hall set (H, S) as per Definition 3.2, by $A = \{\mathrm{X}_i : i \in H\} \cup \{\mathrm{Y}_j : j \in S\}$, if and only if there does not exist any arc in G^* from A to any other SCC.*



(a) Delete unused match (b) New residual graph (c) Partition into **F** and SCCs

Fig. 3.3 Example of propagation using Tarjan's SCC algorithm

Proof All nodes in $(\mathbf{X} \cup \mathbf{Y}) \backslash \mathbf{F}$ are matched, since all unmatched value nodes are in **F**. So A has only matched nodes, which occur in pairs according to Lemma 3.1. Thus $|H| = |S|$ in the theorem, as per Definition 3.2. Arcs in G^* from A to another SCC can be either $(\mathrm{Y}_j, \mathrm{X}_i)$ or $(\mathrm{X}_i, \mathrm{Y}_j)$, where $\mathrm{X}_i \in \mathbf{X}, \mathrm{Y}_j \in \mathbf{Y}$. In the first case X_i is matched to Y_j thus A is trivial according to Lemma 3.1, which doesn't happen by assumption. In the second case $j \in D(x_i)$ and $j \notin S$, so S does not cover the domains of variables in H. Conversely, if the second case doesn't happen then S covers the domains of variables in H, as per Definition 3.2. □

Suppose A is an SCC and $(\mathrm{X}_i, \mathrm{Y}_j) \in G^*$ is an arc from **F** or another SCC to A, that is $\mathrm{Y}_j \in A$. Further suppose there are no arcs in G^* from A to any other SCC, so that A encodes Hall set (H, S), where $i \notin H$ and $j \in S$. Then $x_i = j$ is an unsupported domain value, which we can remove with the explanations proposed by Katsirelos [61] and Rochart et al. [95],

$$\bigwedge_{k \in H} \bigwedge_{\ell \in D \backslash S} [x_k \neq \ell] \rightarrow [x_i \neq j], \tag{3.6}$$

where the LHS simply restates Definition 3.2 in literals, so that if the LHS holds then the values in S, in particular j, are unavailable for any other variable.

Example 3.6 Consider Example 3.4 except with initial support $x_1 = 1, x_2 = 2, x_3 = 3$. Suppose propagation sets $x_1 \neq 4$ as shown in Figure 3.3a. Although this isn't in the current matching, the *alldifferent* propagator wakes up and recalculates G^* as shown in Figure 3.3b, and the set **F** and the SCCs of $(\mathbf{X} \cup \mathbf{Y}) \backslash \mathbf{F}$ as shown in Figure 3.3c. Nothing is reverse-reachable from free value nodes, so **F** is just the only free value node $\{\mathrm{Y}_4\}$. SCCs A and B are a pair of trivial SCCs encoding the vital arc $(\mathrm{X}_1, \mathrm{Y}_1)$, i.e. that $D(x_1) = \{1\}$. SCC C encodes the Hall set (H, S) where $H = \{2, 3\}$ and $S = \{2, 3\}$, that is $D(x_2) \cup D(x_3)$ has size 2. We can remove the arc B → C using this Hall set. The overall domain of the constraint is $D = \{1, 2, 3, 4\}$, so the explanation encodes that $x_2, x_3 \notin D \backslash S = \{1, 4\}$ making S unavailable to x_1,

$$[x_2 \neq 1] \wedge [x_2 \neq 4] \wedge [x_3 \neq 1] \wedge [x_3 \neq 4] \rightarrow [x_1 \neq 2]. \qquad \square$$

On the other hand, suppose A is an SCC and $(\mathrm{X}_i, \mathrm{Y}_j) \in G^*$ is an arc from $\mathbf{F}$ or another SCC to A, and there is an arc in G^* from A to another SCC B. Then A will not be a Hall set until at least the arc from A to B has been removed using B as the explanation (but B has to be a Hall set, etc). In this case A is a predecessor of B in the DAG returned by Tarjan's algorithm. We can propagate the SCCs in any valid topological ordering defined by the DAG, so that all successors of A have been propagated before A. Then arcs from A to other SCCs can be removed with explanation, because those other SCCs *then* encode Hall sets.

We remove arcs from $\mathbf{F}$ to an SCC A using the same method, we just have to ensure that A is safe, e.g. by processing $\mathbf{F}$ last. Also, trivial SCCs always occur in pairs $(\mathrm{X}_i, \mathrm{Y}_j)$ indicating vital arcs where $D(x_i) = \{j\}$, and such a pair can be treated as a non-trivial SCC encoding a singleton Hall set (a single unavailable domain value) in the above. Here we can optionally use the explanations (3.3) instead of (3.6), since they are shorter.

Following Gent et al. [54], we incrementally partition down a branch of the search tree, only re-evaluating partitions that change, and re-merging partitions upon backtracking.

3.6 Feydy-consistent *alldifferent* by decomposition

Instead of designing a monolithic propagator for *alldifferent* which has explanation capability, we can post a decomposition into primitive constraints which already have explanation capability. We showed several decompositions of *alldifferent* in Equations (3.1)–(3.2).

Decompositions with stronger propagation are also available. Feydy and Stuckey [41] propose a decomposition of *alldifferent* specifically for LCG solvers, taking advantage of the literals $[x_i = v]$ and $[x_i \leq v]$ which are already available in an LCG solver as part of the integer variable encoding. Feydy and Stuckey's decomposition is

$$
\begin{aligned}
&\mathit{alldifferent}([x_1, \ldots, x_n]) \equiv \exists c_1, \ldots, c_m \in 0..1, s_0, \ldots, S_m \in 0..n \text{ such that} \\
&\quad s_0 = 0, \\
&\quad s_m = n, \\
&\quad s_i = s_{i-1} + c_i && \forall i \in 1..m, && (3.7) \\
&\quad c_i = \sum_{j \in 1..n} [x_j = i] && \forall i \in 1..m, && (3.8) \\
&\quad s_i = \sum_{j \in 1..n} [x_j \leq i] && \forall i \in 1..m, && (3.9) \\
&\quad \text{where } m \text{ is such that } D(x_j) \subseteq 1..m \; \forall j \in 1..n,
\end{aligned}
$$

noting that $[x_j = i]$ and $[x_j \leq i]$ are taken as the numeric truth values of the literals where $0 = \mathit{false}$ and $1 = \mathit{true}$, and that $c_1, \ldots, c_m, s_0, \ldots, s_m$ are intermediate variables created for the decomposition. The LCG solver can learn on these variables, potentially creating shorter and more reuseable nogoods than those of the monolithic propagators.

The constraints (3.8) implement *alldifferent* with the same propagation strength as the arc-consistent propagator as in Equation 3.2, since $c_i \in 0..1, i \in 1..m$. That is, any literal $[x_j = i] = 1$ makes all the other literals $[x_k = i] = 0, k \neq j$, as we saw in (3.3).

The constraints (3.9) implement a stronger consistency based on Hall intervals. To simplify the analysis, suppose $D(c_i) = 0..1, i \in 1..m$. That is, we aren't sure yet which domain values in the overall domain D of the constraint will be used. Then the constraints (3.7) state that $D(s_i) = \max(i + n - m, 0).. \min(i, n)$, so $D(s_i)$ gets smaller as i approaches either end of the interval $1..m$. Now consider the '$\geq$' side of the constraints (3.9) when i is small (n or less). Substituting $\text{ub}(s_i) = i$ as the the most generous possible value for s_i gives

$$i \geq \sum_{j \in 1..n} [x_j \leq i],$$

that is, suppose i of the literals $[x_j \leq i] = 1$ in this *linear* constraint, then the remaining literals must be 0. Suppose H is the index set such that $[x_j \leq i] = 1, j \in H$. Then $|H| = i$ and we have a Hall interval $(H, 1..i)$. This propagates with an explanation

$$\bigwedge_{j \in H} [x_j \leq i] \rightarrow [x_k > i] \qquad \forall k \in 1..n, k \notin H.$$

This is exactly like the bounds propagator except that Hall intervals must start with 1. Considering the '$\leq$' side of (3.9) gives Hall intervals ending with m.

Now considering the constraints (3.7) more carefully, suppose a variable x_j becomes fixed to i, or suppose i becomes unuseable because $x_j \neq i$ for all variables. Then i is an uninteresting value and the constraint can be rewritten excluding this value, and excluding x_j which takes this value if necessary, to make the overall domain of the constraint $1..m - 1$ with the interesting values contiguous again. Then the above analysis holds.

However, the consistency is slightly better than bounds because the Feydy and Stuckey decomposition knows about domain holes via the constraints (3.7), so it will e.g. see a Hall interval of size 3 if there are 3 variables taking values 1..4 and the value 2 is not available. This is potentially valuable for *alldifferent* constraints over sparse domains.

Overall, provided the decomposition is not too large, the decomposition approach can be faster than a monolithic propagator, since no advanced propagation algorithm such as Algorithm 3.1 is required. Globality is lost since the primitive constraints don't know the deeper consequences of *alldifferent*, but this can often be recovered using learning, especially if the decomposition uses its own variables as does Feydy and Stuckey's.

3.7 Experiments

In these experiments we evaluate the state-of-the-art LCG solver *LCG-glucose*, using the *alldifferent* propagators with arc, bounds and domain consistency extended to explain themselves

in an LCG solver, and the Feydy-consistent *alldifferent* decomposition. We compare *LCG-glucose* with and without learning, against the state-of-the-art non-learning solver *Gecode* 4.4.0 [52], which has a comprehensive suite of *alldifferent* propagators implementing the same consistency levels, but without explanation capability.

We aim to test which propagator is appropriate for each problem, and whether learning provides an additional benefit over just choosing the appropriate propagator. We expect *LCG-glucose* without learning to have somewhat more overhead than *Gecode*, since it still uses a Boolean model of domains, even though it doesn't use conflict analysis or nogood recording. We expect that on appropriate problems, enabling learning will pay for this overhead and the additional overhead of conflict analysis and nogood propagation.

We run on Intel dual processor × quad core Xeon E5405 nodes at 2.0 GHz with 16 Gb RAM per node. All solver executables are single-threaded and each core runs independently as a virtual single-threaded CPU, but we never schedule two different solvers simultaneously on the same physical node. Timeouts are 600s and each core uses 1.5 Gb RAM.

We examine 14 different well-known problems using *alldifferent*, of which 9 are satisfaction and 5 are optimization problems. For each problem, if possible we generate up to 25 instances, with appropriate difficulty level so that about half of the problems can be solved on average. Some of the problems don't admit instance data other than the dimension of the problem, and these use a more restricted instance set appropriate to the problem. We have made available on our website `http://people.eng.unimelb.edu.au/pstuckey/alldifferent`, high level models and data in MiniZinc [80] format.

All solvers use a *first_fail* i.e. smallest-domain-first search strategy. Using a dynamic strategy like *first_fail* is essential for these problems, as compared with a static *input_order* strategy. However, dynamic search can introduce noise, particularly on satisfaction problems, which don't necessarily explore the entire search space. Therefore, on each problem, each solver tackles each of the up to 25 instances 10 times with a different search order on each attempt, based on randomly permuting the *first_fail* tie-break ordering.

We generate 10 permutations of each FlatZinc [11] model a priori, each having a different tie-break order. The runtime of a solver on an instance is taken as the median of runtimes over the 10 permutations (with 600 s representing a timeout). For each problem and solver we show the geometric mean of this runtime over the up to 25 instances in the set.

The problems tackled in this experiment are as follows.

1. Costas Array (Costas; 11 instances; satisfaction), with $n \in 10..20$. An order-n Costas array is a permutation on $1..n$ such that the distances in each row of the triangular difference table are distinct.
2. Debruijn Sequence (Debruijn; 7 instances; satisfaction), with $k = 2, n \in 6..8; k = 3, n \in 4..5; k = 4, n \in 3..4$. A Debruijn sequence is a minimal sequence to type for testing all possible code sequences of length n on a device with k keys, where there is no requirement to press Enter after each try.

3. Golomb Rulers (Golomb; 5 instances; satisfaction), prob006 in CSPLib [53], with $n \in 8..12$. A Golomb ruler has n irregularly spaced marks such that the integer distances between all pairs of marks are distinct.
4. Kakuro (Kakuro; 16 instances; satisfaction), on a 10×10 up to 17×17 grid with 40% or 60% utilization. Kakuro is a kind of crossword where each 'word' is a permutation of a subset of 1..9 summing to a given value. We use the grid generator at `http://www.perlmonks.org/?node_id=550884`. We use Simonis's redundant *alldifferent_sum* constraints, but not his *interact* constraints [104].
5. Knights Tour (Knights; 8 instances; satisfaction), of length $50, 52, \ldots, 64$, on an 8×8 chessboard. This problem is finding a circular tour of the specified length using legal knight moves starting in one corner of a chess board.
6. Linear-to-Program (L2P; 5 instances; optimization), from MiniZinc Challenge 2013 [30]. This problem is finding a shortest program to compute a linear combination.
7. Langford's Number (Langford; 22 instances; satisfaction), prob024 in CSPLib [53], with $k = 2, n \in 9..24; k = 3, n \in 3..18; k = 4, n = 24$. This problem is arranging k sets of numbers $1..n$, so that each appearance of a number m is m positions from the last.
8. Minimization of Open Stacks (MOSP; 15 instances; optimization), from MiniZinc benchmarks, `https://github.com/MiniZinc/minizinc-benchmarks`. This problem is given a set of customers and orders, determine the sequence in which products should be manufactured to minimize the maximum number of open stacks, i.e. the maximum number of customers whose orders are simultaneously active. We choose instances by ranking those available by *customers* $\times$ *orders*, and taking every 3rd instance.
9. Perfect 1-Factorizations (P1F; 10 instances; optimization), from MiniZinc Challenge 2015 [30], with $n \in 10..20$. This problem is partitioning a complete graph into complete matchings subject to a Hamiltonian circuit constraint.
10. Photo (Photo; 16 instances; optimization), from MiniZinc benchmarks, with $n \in 10..13$. This problem is arranging n people in a photo, respecting as many of their preferences to stand next to a friend as possible. We generate challenging instances with 3 preferences per person and a popularity model based on rolling two dice.
11. Quasigroup Completion (QCP; 22 instances; satisfaction), with $n \in 25..35$. This problem is completing a partially filled $n \times n$ Latin Square [55]. We use Gomes's *lsencode* generator at `http://www.cs.cornell.edu/gomes/SOFT/lsencode-v1.1.tar.Z`, eliminating easy instances by making 100 attempts to solve by propagation using *LCG-glucose* with arc-consistency and branching but no backtracking.
12. Quasigroup Existence (Quasigrp; 24 instances; satisfaction), prob003 in CSPLib [53], on *QG3..7* with $n \in 8..10$. This problem is creating an $n \times n$ Latin Square with various mathematical properties. We use Colton and Miguel's redundant constraints [28].
13. Social Golfer (SGolfer1/2; 15 instances; satisfaction), prob010 in CSPLib [53], with $g, s, w \in 2..8$. This problem is finding a roster for n golfers to play each other in g

groups of s over w weeks, such that no golfer plays in the same group as another golfer twice, where $n = gs$. We rank instances by difficulty based on the number of possible pairs of golfers, eliminate easy instances by making 100 attempts to solve by propagation using *LCG-glucose* with arc-consistency and branching but no backtracking, and take the easiest 15 remaining. SGolfer1 investigates *alldifferent* within weeks, using arc-consistency for *alldifferent* across weeks, and SGolfer2 vice versa.

14. Superblock Instruction Scheduling (Superblk; 25 instances; optimization). This problem is finding the fastest schedule for a 'superblock' of CPU instructions subject to dependencies and jump-taken probabilities, for single-issue CPUs, which use *alldifferent*. We translate Malik et al.'s single-issue model and instances [68] to MiniZinc. We choose one instance from each set, by ranking the available instances by decreasing file size, skipping the top 20% of instances, and then eliminating easy instances by making 100 attempts to solve by propagation using *LCG-glucose* with bounds-consistency and branching but no backtracking, until we find a suitable instance.

Tables 3.1 and 3.2 show the results of this experiment. We see that Costas, Debruijn, Golomb, Langford, Photo and Quasigrp did not benefit from learning, and these problems also didn't benefit significantly from higher consistency levels than arc-consistent. QCP is a more complex case since Feydy-consistency and domain-consistency do well with and without learning respectively, thus QCP admits several different solving approaches. For problems that didn't benefit from learning, in general *Gecode* was somewhat faster, an exception is Photo, for which *LCG-glucose* has stronger consistency than *Gecode* for one of the constraints and therefore solves these problems faster even without learning.

The remaining problems benefited enormously from learning, closing many extra instances, except L2P which solved the same instances in 40% of the runtime. Of these, Kakuro solved best with arc-consistency, and the remaining problems benefited from some form of stronger consistency. This shows that LCG solvers should implement higher consistency levels for *alldifferent* than just arc-consistency, despite the added complexity of adding explanation capability to the propagators or providing better decompositions.

On the problems which benefit from learning, bounds-consistent *alldifferent* with learning performed quite well, giving clearly the best results on Knights, SGolfer2 and Superblk. The bounds-consistent *alldifferent* and *gcc* propagators [65, 89] were designed in conjunction with the Superblk model [68], so we know these problems benefit from bounds consistency, but adding explanations gives a further *huge* improvement on these problems.

Domain-consistent *alldifferent* with learning was disappointing. Without learning, domain-consistency gives the best results on Quasigrp and SGolfer1/2, but this didn't carry over to learning, showing that learning significantly changes the tradeoffs for propagation. On the other hand, domain-consistent *alldifferent* was reasonably competitive on all problems except Superblk, showing that it can pay for at least some of its propagation overhead.

solver	algorithm	Costas(11) TO	Costas(11) time s	Debruijn(7) TO	Debruijn(7) time s	Golomb(5) TO	Golomb(5) time s	Kakuro(16) TO	Kakuro(16) time s	Knights(8) TO	Knights(8) time s
not learning:											
Gecode	value	**3**	**7.30**	**0**	**0.02**	1	9.42	10	54.43	8	600.00
	bounds	3	7.48	**0**	**0.02**	**1**	**9.40**	10	54.43	8	600.00
	domain	4	7.41	0	0.07	1	11.52	10	53.49	8	600.00
	Feydy	4	20.19	0	1.97	2	52.41	10	88.61	8	600.00
LCG-glucose	value	4	13.10	0	0.05	2	48.15	10	54.46	8	600.00
	bounds	4	16.85	0	0.06	2	36.95	14	348.52	8	600.00
	domain	4	13.43	0	0.09	2	38.68	11	102.09	8	600.00
	Feydy	4	20.57	0	2.00	3	143.47	10	72.93	8	600.00
learning:											
LCG-glucose	value	4	14.15	0	0.06	2	42.50	**2**	**1.29**	4	81.34
	bounds	4	17.15	0	0.06	2	29.38	4	4.63	**4**	**67.04**
	domain	4	14.27	0	0.09	2	35.11	6	5.00	4	98.53
	Feydy	4	24.40	0	2.03	2	103.73	2	1.39	4	125.31

solver	algorithm	L2P(5) TO	L2P(5) time s	Langford(22) TO	Langford(22) time s	MOSP(15) TO	MOSP(15) time s	P1F(10) TO	P1F(10) time s	Photo(16) TO	Photo(16) time s
not learning:											
Gecode	value	1	52.47	5	1.62	12	281.86	8	220.03	4	91.82
	bounds	1	53.40	**5**	**1.61**	12	281.78	8	217.91	4	91.51
	domain	1	52.89	5	1.63	12	277.92	8	218.81	4	84.37
	Feydy	1	52.60	5	2.97	12	285.94	8	238.33	8	143.34
LCG-glucose	value	2	269.83	5	2.02	12	262.63	9	326.06	**4**	**36.64**
	bounds	2	270.43	5	2.04	12	264.01	7	159.81	7	137.49
	domain	2	270.27	5	2.03	12	260.89	8	200.73	4	42.10
	Feydy	2	269.00	5	3.07	12	266.23	8	219.87	4	40.61
learning:											
LCG-glucose	value	1	19.53	5	2.26	10	110.49	4	25.25	6	97.44
	bounds	**1**	**19.13**	5	2.27	10	123.64	4	23.92	9	190.75
	domain	1	19.82	5	2.31	10	111.56	**4**	**23.46**	7	99.51
	Feydy	1	19.99	5	3.57	**10**	**98.89**	4	44.78	7	108.09

Table 3.1 *alldifferent* experiment (continues next page)

With learning, comparing the bounds-consistent with the domain-consistent results shows that the bounds-consistent propagator due to its simplicity is probably executing faster, and also that the extremely compact nogoods produced by the bounds-consistent propagator are probably much more efficient to propagate, as well as being more reuseable, than the large and cumbersome nogoods produced by the domain-consistent propagator.

The Feydy-consistent *alldifferent* decomposition was also a surprise good performer with learning, giving the best solutions for MOSP, QCP and SGolfer1, despite its simplicity. The

solver	algorithm	QCP(22) TO	QCP(22) time s	Quasigrp(24) TO	Quasigrp(24) time s	SGolfer1(15) TO	SGolfer1(15) time s	SGolfer2(15) TO	SGolfer2(15) time s	Superblk(25) TO	Superblk(25) time s
not learning:											
Gecode	value	22	600.00	4	2.60	8	31.42	8	31.42	7	2.21
	bounds	22	600.00	4	2.60	8	31.23	8	31.73	7	2.19
	domain	4	6.80	**4**	**2.35**	7	28.17	7	20.32	7	3.12
	Feydy	9	56.46	4	3.32	8	100.56	8	98.35	8	18.87
LCG-glucose	value	22	600.00	4	3.31	8	85.61	8	85.61	25	600.00
	bounds	22	600.00	6	10.00	9	148.50	6	36.76	7	3.50
	domain	5	21.64	7	8.44	8	38.02	8	46.77	8	6.99
	Feydy	8	49.73	4	3.50	8	114.78	8	116.94	7	8.73
learning:											
LCG-glucose	value	13	191.84	5	3.84	8	48.06	8	48.06	25	600.00
	bounds	13	292.39	4	4.27	7	49.23	**6**	**31.63**	**0**	**0.11**
	domain	4	15.94	7	8.44	8	34.47	8	45.12	5	1.65
	Feydy	**3**	**18.49**	4	3.56	**6**	**76.99**	6	79.94	0	1.08

Table 3.2 *alldifferent* experiment (from previous page)

same decomposition didn't perform nearly as well on the same problems without learning, showing that learning can discover some of the stronger consequences of the decomposition. The Feydy decomposition was usually the slowest on *Gecode* which doesn't have the literals used by the decomposition and has to create them through reification. This shows that these literals are cheaper in an LCG solver.

3.8 Conclusions

The experiments demonstrate that LCG changes the tradeoffs for propagation. Whereas expensive propagation algorithms can pay for themselves in CP where the reduction in search is considerable, they have more difficulty paying for themselves in LCG.

In some cases the expensive propagation algorithm was helpful in LCG, although less dramatically so than in CP. In other cases the expensive propagation algorithm was actually a disadvantage in LCG. It looks as if the expensive propagator is either not paying for its propagation cost in reduced search (i.e. reducing search but increasing solving time), or is actually increasing search by producing longer and less reuseable nogoods.

The experiments also demonstrate that LCG changes the tradeoffs for decomposition. Whereas decomposition is usually a big disadvantage in CP where the lost globality through decomposition leads to extra search, it is less of a disadvantage in LCG.

In some cases the decomposition was also unhelpful in LCG, although less dramatically so than in CP. In other cases the decomposition was actually an advantage in LCG. It looks as if the decomposition is producing better nogoods, possibly because of the intermediate

variables the decomposition creates. LCG can learn on the intermediate variables, and also the intermediate variables reduce duplication leading to more concise nogoods.

Taken together, the above results show that LCG can discover the stronger consequences of *alldifferent* through learning, and thereby wholly or partially recover the globality lost through using a cheaper propagator for *alldifferent* or indeed a decomposition.

Overall, the best *alldifferent* propagation algorithm was the bounds-consistent propagator with explanations, as it is relatively cheap to execute and produces very compact explanations. The next best approach was the Feydy-consistent decomposition, which has slightly stronger propagation in some situations, and propagates efficiently if not too large.

LCG solvers should still offer a range of propagation approaches to *alldifferent*, since some problems benefit from dedicated propagators even in LCG. In such cases, e.g. Superblk [68], the search reduction due to higher consistency and the search reduction due to learning are orthogonal, and huge benefits can be seen using both together.

Chapter 4

Explaining Network Flow Constraints

4.1 Introduction

The original generic flow-based constraint for CP, proposed by Bockmayr et al. [19], is: Given a collection of nodes, arcs, costs, and variables, flow conservation must hold, flow bounds must not be exceeded, and the overall cost of the flow must be assigned to a variable. This occurs in transportation, logistics, mining, mixing problems, etc.

Flow-based constraints were initially implemented by Bockmayr et al. in the earlier constraint system *CHIP* [19]. A more modern implementation of flow-based constraints is by Steiger et al. in *JaCoP* [106]. Another noteworthy application of flow-based constraints is 'behind the scenes' in specific propagators for global constraints such as *alldifferent* [90], *gcc* [91] or *sliding_sum* [67], which can be expressed as flow.

LCG solvers did not have a generic flow-based constraint until the work described in this chapter, a preliminary version[1] of which we presented in CPAIOR '12. This is because the existing generic flow-based propagators for traditional CP cannot be used in an LCG solver without being extended to *explain* themselves by clauses.

Rochart in his thesis had discussed generating explanations for flow-based constraints without costs [94]. Linear Programming (LP) solvers also routinely provide explanations for conflicts. Since Minimum Cost Flow (MCF) is expressible as an LP, we can use the same approach to explain flow-based constraints with costs. Davey et al. [31] and Achterberg [2]

[1]Downing, N., Feydy, T., Stuckey, P.J.: Explaining flow-based propagation. In: N. Beldiceanu, N. Jussien, É. Pinson (eds.) *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems: 9th International Conference, CPAIOR 2012, Nantes, France, May 28 – June 1, 2012. Proceedings*, pp. 146–162. Springer Berlin Heidelberg (2012)

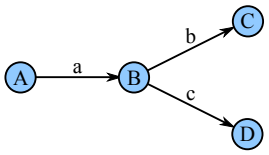


Fig. 4.1 Example of a directed network

also took a similar approach to explaining LP conflicts, but they used the explanations for cut-generation only, which is much weaker than clausal propagation.

We give an experimental evaluation of the new explanation-capable flow-based propagators, with and without costs. We implemented explanations for all of the existing propagation (filtering) methods supported by traditional CP solvers for the generic flow constraint [19, 106] or flow-based propagators for specific constraints such as Régin's domain-consistent *alldifferent* [90]. We also try an innovative propagation method based on caching and propagating the explanation certificates as redundant *linear* constraints.

We show that a dedicated flow-based constraint is absolutely essential for problems involving flow, since decomposition to *linear* constraints (the only approach available in LCG prior to our work) loses much too much globality, even in an LCG solver which can recover some globality by learning. We also show that the tradeoffs for propagation are different between LCG and traditional CP solvers, in terms of the time spent propagating and the strength of the resulting filtering, versus the expected reduction in search.

4.2 Directed networks and flows

We define directed networks in a similar manner to Ford and Fulkerson [46], but with some added generality whereby arcs are individually identified, so that parallel arcs and self-loop arcs may be considered. We require this feature for modelling reasons.

Definition 4.1 A *directed network* $G = [N; A; \mathcal{T}; \mathcal{H}]$ consists of a set $N = \{\mathrm{A}, \mathrm{B}, \ldots\}$ of nodes which we have named with uppercase alphabetic characters for convenience in referring to diagrams etc, a set $A = \{\mathrm{a}, \mathrm{b}, \ldots\}$ of arcs similarly named in lowercase, and functions $\mathcal{T} : A \to N$ (tail) and $\mathcal{H} : A \to N$ (head) describing the connectivity of arcs.

Graphically we show G as a collection of circles (nodes) labelled $n \in N$, connected by directed arrows (arcs) labelled $a \in A$ drawn from node $\mathcal{T}(a)$ to node $\mathcal{H}(a)$.

Example 4.1 Let $N = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}\}$, $A = \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$, $\mathcal{T} = \{\mathrm{a} \mapsto \mathrm{A}, \mathrm{b} \mapsto \mathrm{B}, \mathrm{c} \mapsto \mathrm{B}\}$, $\mathcal{H} = \{\mathrm{a} \mapsto \mathrm{B}, \mathrm{b} \mapsto \mathrm{C}, \mathrm{c} \mapsto \mathrm{D}\}$. The resulting directed network $[N; A; \mathcal{T}; \mathcal{H}]$ is shown in Figure 4.1. □

We adopt the following notation: Where $X \subseteq N$ we write $\overline{X}$ for $N \backslash X$. We use preimage notation $\mathcal{G}^{-1}(X)$ for the set $\{a \in A : \mathcal{G}(a) \in X\}$. We use the shorthand (X, Y) for the set

$\mathcal{T}^{-1}(X) \cap \mathcal{H}^{-1}(Y)$, that is the set of arcs crossing from nodes in X to nodes in Y. Any of these X or Y may be singletons in which case we just write the node directly. In an adaptation of Ford and Fulkerson's notation [46] we write $g(X, Y)$ for $\sum_{a \in (X,Y)} g(a)$.

We will primarily discuss *circulations*, which are presented by Ford and Fulkerson [46, §3] as an extension of their main theory which concerns *static flows* or *maximum flows*. We prefer circulations because they embed more naturally into a CP solver, given that they (i) are satisfaction problems, and (ii) accommodate lower bounds on flow values. We adapt the following definitions about circulations from Ford and Fulkerson.

Definition 4.2 Let $G = [N; A; \mathcal{T}; \mathcal{H}]$ be a directed network and $l, u : A \to \mathbb{R}$ be lower and upper bound functions respectively, such that $l(a) \leq u(a)$ for all $a \in A$. A *feasible circulation* in G is a flow function $f : A \to \mathbb{R}$ satisfying

$$f(x, N) - f(N, x) = 0 \qquad \forall x \in N, \tag{4.1}$$

$$l(a) \leq f(a) \leq u(a) \qquad \forall a \in A. \tag{4.2}$$

Theorem 4.1 *Given G, l, and u as above, a feasible circulation exists if and only if*

$$u(X, \overline{X}) - l(\overline{X}, X) \geq 0 \qquad \forall X \subseteq N.$$

Note that in the above we have removed the restriction of Ford and Fulkerson that bounds and/or flows be nonnegative. We do not re-prove Theorem 4.1 under these conditions because Berge and Ghouila-Houri [15, pp 195–196] show how to do so, at least in principle.

The following definitions are useful for finding feasible circulations.

Definition 4.3 Let a *path* $\pm a_1, \pm a_2, \ldots, \pm a_n$ be a sequence of arcs, with associated direction $+$ or $-$, connected via intermediate nodes $x_1, x_2, \ldots x_{n+1}$, such that

$$(x_i, x_{i+1}) = \begin{cases} (\mathcal{T}(a_i), \mathcal{H}(a_i)), \text{ direction of } a_i \text{ is } + \\ (\mathcal{H}(a_i), \mathcal{T}(a_i)), \text{ direction of } a_i \text{ is } - \end{cases} \qquad \forall i \in 1..n.$$

Definition 4.4 Let G be a directed network, and $f, l, u : A \to \mathbb{R}$ a circulation, lower, and upper bound function respectively. Suppose the circulation f obeys the flow conservation conditions (4.1) but not necessarily the bounds conditions (4.2). The *residual graph* $G^* = [N; A^*; \mathcal{T}^*; \mathcal{H}^*]$ of such a circulation is defined by

$$A^* = \{+a \ : \ a \in A, f(a) < u(a)\} \cup \{-a \ : \ a \in A, f(a) > l(a)\}$$
$$\mathcal{T}^* = \{+a \mapsto \mathcal{T}(a) \ : \ +a \in A^*\} \cup \{-a \mapsto \mathcal{H}(a) \ : \ -a \in A^*\}$$
$$\mathcal{H}^* = \{+a \mapsto \mathcal{H}(a) \ : \ +a \in A^*\} \cup \{-a \mapsto \mathcal{T}(a) \ : \ -a \in A^*\}.$$

The residual graph indicates which f may legally increase while maintaining, or at least not further violating, the flow bounds conditions (4.2). We say 'increase' because if some f may

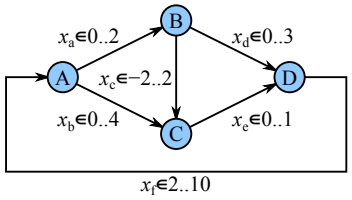


Fig. 4.2 Example of a circulation problem

legally decrease, the corresponding residual arc is reversed from its original orientation, so it 'negatively increases'. We use residual graphs in Sections 4.4.1 and 4.4.3.

4.3 Constraint for circulations

We define the constraint $circulation(G, \mathbf{x})$ for use in CP solvers, as follows,

$$circulation([N; A; \mathcal{T}; \mathcal{H}], [x_a]_{a \in A}) \equiv \bigwedge_{u \in N} \left(\sum_{a \in (u,N)} x_a - \sum_{a \in (N,u)} x_a = 0 \right).$$

Then asking the CP solver to solve a feasible circulation problem consists in setting up variables $x_\mathrm{a}, x_\mathrm{b}, \ldots$ with appropriate bounds conditions as in (4.2) which will be enforced by the CP solver, and then passing these already-bounded variables to the *circulation* constraint, which will enforce what is effectively the flow conservation condition (4.1).

Example 4.2 Given the network G of Figure 4.2 which defines the flow balance conditions

$$\begin{aligned} &\text{(A)} \quad x_\mathrm{a} + x_\mathrm{b} - x_\mathrm{f} = 0 && \text{(B)} \; -x_\mathrm{a} + x_\mathrm{c} + x_\mathrm{d} = 0 \\ &\text{(C)} \; -x_\mathrm{b} - x_\mathrm{c} + x_\mathrm{e} = 0 && \text{(D)} \; -x_\mathrm{d} - x_\mathrm{e} + x_\mathrm{f} = 0, \end{aligned}$$

the following Constraint Program adds the necessary flow bounds conditions to encode a complete circulation problem into CP using the *circulation* constraint,

$$\begin{aligned} &\text{find } x_\mathrm{a} \in 0..2, x_\mathrm{b} \in 0..4, x_\mathrm{c} \in -2..2, x_\mathrm{d} \in 0..3, x_\mathrm{e} \in 0..1, x_\mathrm{f} \in 2..10 \\ &\text{such that } circulation(G, [x_\mathrm{a}, x_\mathrm{b}, x_\mathrm{c}, x_\mathrm{d}, x_\mathrm{e}, x_\mathrm{f}]). \end{aligned}$$

□

4.4 Propagator for circulations

The propagation algorithm for *circulation* consists of two phases:

(i) Checking: Does the constraint have any solution to $\mathbf{x}$ given the current bounds $\mathrm{lb}(x_a)$ and $\mathrm{ub}(x_a)$ for a in A? If so, the propagator proceeds to the propagation phase. If not, the propagator informs the propagation engine that the search must backtrack.

Algorithm 4.1 Breadth-first labelling algorithm to check and repair circulations

Step 1. Start with any integral valued f that satisfies the flow conservation conditions (4.1). For example, $f = 0$ will do.
Step 2. Create a list Q which will be partitioned at position i into nodes already visited and nodes to visit. Initially $Q = []$ and $i = 1$.
Step 3. If bound conditions (4.2) are satisfied then return f as a feasible circulation. Otherwise locate an arc a for which one of the flow bounds conditions is violated, and then:

(a) If $f(a) < l(a)$: Set $s \leftarrow \mathcal{H}(a), t \leftarrow \mathcal{T}(a)$. Label s with $[+a, \epsilon(s) = l(a) - f(a)]$, and then if $s = t$ then go to step 6, else append s to Q and continue.
(b) If $f(a) > u(a)$: Set $s \leftarrow \mathcal{T}(a), t \leftarrow \mathcal{H}(a)$. Label s with $[-a, \epsilon(s) = f(a) - u(a)]$, and then if $s = t$ then go to step 6, else append s to Q and continue.

Step 4. While $i < |Q|$, set $x \leftarrow Q[i]$, $i \leftarrow i + 1$, and visit x as follows:

(a) For b in $\mathcal{T}^{-1}(x)$ with $f(b) < u(b)$: Set $y \leftarrow \mathcal{H}(b)$. If y is unlabelled: Label y with $[+b, \epsilon(y) = \min(\epsilon(x), u(b) - f(b))]$, and then if $y = t$ then go to step 6, else append y to Q and continue.
(b) For b in $\mathcal{H}^{-1}(x)$ with $f(b) > l(b)$: Set $y \leftarrow \mathcal{T}(b)$. If y is unlabelled: Label y with $[-b, \epsilon(y) = \min(\epsilon(x), f(b) - l(b))]$, and then if $y = t$ then go to step 6, else append y to Q and continue.

Step 5. No breakthrough. Return Q as an infeasibility certificate.
Step 6. Breakthrough. Construct a path P, initially empty, as follows. Set $x \leftarrow t$. Node x is labelled with either $+b$ such that $\mathcal{H}(b) = x$ or $-b$ such that $\mathcal{T}(b) = x$. Prepend this label to P. If it is $+b$ then set $x \leftarrow \mathcal{T}(b)$ or if it is $-b$ then set $x \leftarrow \mathcal{H}(b)$. Repeat until $x = s$.
Step 7. Augment along path P (which is in fact a cycle) by $\epsilon(t)$, that is for each arc $+b$ in P set $f(b) \leftarrow f(b) + \epsilon(t)$ and for each arc $-b$ in P set $f(b) \leftarrow f(b) - \epsilon(t)$. Return to step 2.

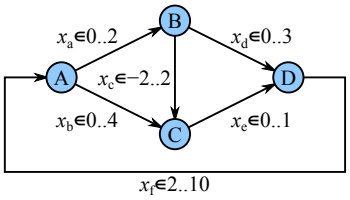
(ii) Pruning: Can some bounds be pruned? That is, for some a in A could we increase $\text{lb}(x_a)$ or decrease $\text{ub}(x_a)$ without cutting off any of the current family of solutions to $\mathbf{x}$? This will depend on the structure of the network and some or all of the remaining bounds $\text{lb}(x_b)$ and/or $\text{ub}(x_b)$ for all $b \neq a$. For any such cases that can readily be identified, the propagator informs the propagation engine of the pruned bounds.

In either case, the propagator must justify its action (claiming infeasibility or pruning) by means of explanations. For the purpose of this discussion, an explanation is a globally true redundant *linear* constraint $c_1x_1 + \ldots + c_nx_n \geq d$ where c, d are constant and x are CP variables. The constraint is redundant because it is a consequence of *circulation*.

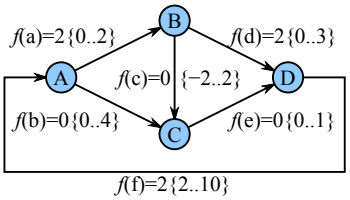
We now examine the steps and the associated explanations in more detail.

4.4.1 Propagator for circulations – checking

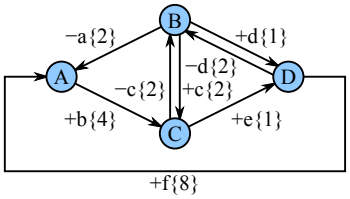
Checking a *circulation* constraint is finding a *support* for the constraint, that is a valuation $f(a) \in D(x_a)$ for the arcs $a \in A$ such that flow conservation (4.1) holds. We do this by setting bounds functions $l(a) = \text{lb}(x_a), u(a) = \text{ub}(x_a)$, thus simplifying $D(x_a)$ to an interval



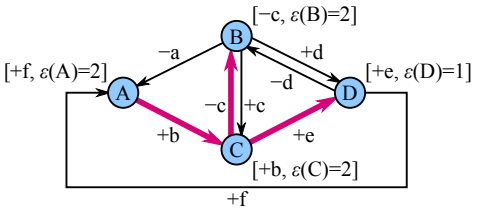
(a) Example of a *circulation* constraint



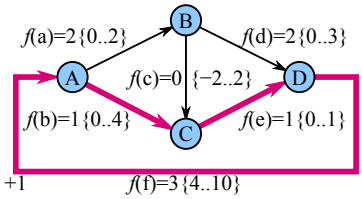
(b) Example resident solution, or support



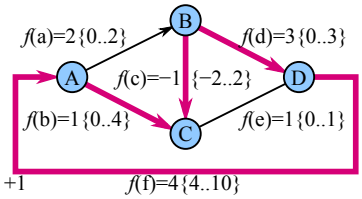
(c) Residual graph G^* of resident solution



(d) Breadth-first search in G^* for a path A→D



(e) First augmenting cycle, shows G not G^*



(f) Second augmenting cycle, new support

Fig. 4.3 Example of checking a circulation using Algorithm 4.1

domain, and applying Ford and Fulkerson's labelling algorithm [46, §3], with a breadth-first search order, to obtain a feasible circulation.

Algorithm 4.1 shows Ford and Fulkerson's algorithm adapted to our definitions. Since the propagator executes every time $\mathrm{lb}(x_a)$ or $\mathrm{ub}(x_a)$ changes, for efficiency we maintain the support $f(a)$ from the last execution, initally set to $f(a) = 0$ for all $a \in A$ as noted in step 1, and try to repair it in order to find a new support for the constraint.

Therefore, on entry to the repair algorithm we have $f(a)$ obeying flow conservation (4.1), but not necessarily flow bounds (4.2). The algorithm requires the residual graph G^*, which is a graphical (or connectivity-based) representation of which arcs a can legally increase i.e. $f(a) < u(a)$, or can legally decrease i.e. $f(a) > l(a)$. The algorithm repeatedly finds an arc a which is out of bounds, performs a breadth-first search in the residual graph, starting at $\mathcal{H}(+a)$ and looking for $\mathcal{T}(+a)$, and augments along that cycle if one exists.

Example 4.3 Consider solving the Constraint Program of Example 4.2, with some arbitrary extra side constraints. Figure 4.3a shows the flow network and initial domains of the variables. The initial supports are $f(a), a \in A$ as shown in Figure 4.3b, with the flow bounds $l(a), u(a)$ reproduced from the domains $D(x_a)$ of Figure 4.3a in curly braces.

The residual graph G^* of this solution is as shown in Figure 4.3c. Now suppose some other propagator eliminates the values 2 and 3 from $D(x_\mathrm{f})$. The *circulation* propagator wakes up and tries to find a new support. Now $l(\mathrm{f}) = \mathrm{lb}(x_\mathrm{f}) = 4$ and $f(\mathrm{f})$ in the current resident solution violates flow bounds (4.2), being too low, and needs to increase.

We execute Algorithm 4.1 which tries to augment the residual arc +f. We start at node A which is the head of arc +f, and we label node A as reachable via +f and admitting 2 extra units of flow (enough to make $f(\mathrm{f}) = l(\mathrm{f})$ and satisfy flow bounds). We visit nodes in the order ACBD labelling the arc used to reach each node and the flow admitted (the minimum of the capacity to the previous node on the path and the capacity of the arc).

At node D we have found the tail of arc +f and hence a cycle. The cycle is reconstructed working backwards via the labels as ACDA, where the visit to node B was a dead end (on this iteration) and is ignored, not affecting the path or its capacity. Expressed as a path per Definition 4.3, the cycle is [+b, +e, +f]. The capacity is limited by arc +e to 1.

Augmenting as shown in Figure 4.5e gives a new solution with $f(\mathrm{f})$ still in violation. Repeating the procedure as shown in Figure 4.5f finds a new support for the constraint. □

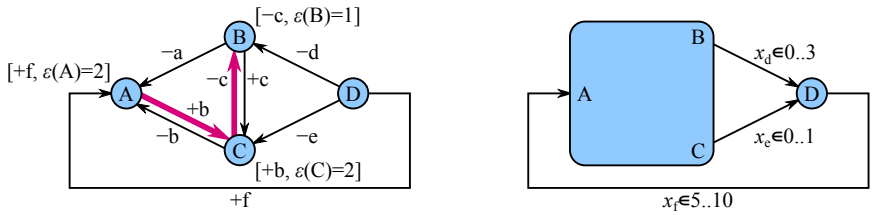
If no new support exists, the CP solver must backtrack. The *circulation* propagator must provide an explanation in order to claim failure and initiate backtracking. The explanation is a redundant *linear* constraint which is obviously infeasible, allowing a clause to be derived from it (using the *linear* propagator as a subroutine), in which all literals are *false* in the current subproblem. The *false* clause is then handed over to the CP solver, which submits it for conflict analysis and then records a clausal nogood prior to backtracking.

We derive the explanation from the infeasibility certificate constructed by Algorithm 4.1 at step 5, as proposed by Rochart et al. [95] and Katsirelos [61] for the restricted case of the *alldifferent* constraint. Rochart et al. went so far as to provide an implementation for *alldifferent* [95], and also briefly discussed in his thesis [94] the case of general flow networks that we consider here, though without implementation or experiments.

The set Q of nodes visited by Algorithm 4.1 defines the boundary of the residual graph, giving a counterexample to Theorem 4.1 in the form of a cut in the graph which cannot support the required flow needed for feasibility. To construct the explanation we sum the the node balance equations over the nodes in Q and then relax '=' to '≥' as follows,

$$\sum_{a \in (Q, \overline{Q})} x_a - \sum_{a \in (\overline{Q}, Q)} x_a \geq 0. \tag{4.3}$$

We now show that if the checking phase fails to find a new support for *circulation*, then the above explanation suffices to prove the lack of any support.



(a) Breadth-first search fails to find a path A→D in G^* (b) Evidence of contradiction in nodes ABC of G

Fig. 4.4 Example of an infeasibility certificate using Algorithm 4.1

Theorem 4.2 *If Algorithm 4.1 fails at step 5 returning an infeasibility certificate as a set of nodes Q, the explanation* (4.3)*, constructed by summing the node balance constraints over the nodes in Q and relaxing '=' to '≥', is obviously infeasible in the current subproblem.*

Proof The *linear* propagator checks the explanation (4.3) by substituting the most generous possible values from $D(x_a)$ so as to maximize the left-hand side, that is, it checks

$$\sum_{a\in(Q,\overline{Q})} \mathrm{ub}(x_a) - \sum_{a\in(\overline{Q},Q)} \mathrm{lb}(x_a) \geq 0,$$

or simply $u(Q,\overline{Q}) - l(\overline{Q},Q) \geq 0$. We show that this is *false* because $u(Q,\overline{Q}) < l(\overline{Q},Q)$.

By construction Q could not be enlarged by following arcs of the residual graph G^*. That is $f(a) \geq u(a)$ for all a in $(Q,\overline{Q})$ and $f(a) \leq l(a)$ for all a in $(\overline{Q},Q)$. Further since f by assumption obeys flow conservation (4.1) one has $f(Q,\overline{Q}) = f(\overline{Q},Q)$. Then

$$u(Q,\overline{Q}) \leq f(Q,\overline{Q}) = f(\overline{Q},Q) \leq l(\overline{Q},Q). \tag{4.4}$$

Equality in (4.4) would require $u(Q,\overline{Q}) = f(Q,\overline{Q})$ and $f(\overline{Q},Q) = l(\overline{Q},Q)$. But this doesn't happen because, referring to step 3, at least one arc must be in violation of flow bounds. □

Example 4.4 Continuing Example 4.3, suppose the current support is as in Figure 4.3f, and some other propagator eliminates the value 4 from $D(x_\mathrm{f})$. Then $l(\mathrm{f}) = \mathrm{lb}(x_\mathrm{f}) = 5$ and the *circulation* propagator tries to increase $f(\mathrm{f})$ by completing a path from A to D.

Breadth-first search in the residual graph G^* proceeds starting at node A, as shown in Figure 4.4a. No path exists to D and the search fails after visiting and labelling nodes ABC. Merging A, B and C to form a super-node as in Figure 4.4b clearly shows the problem: The flow entering ABC must be at least $\mathrm{lb}(x_\mathrm{f}) = 5$, whereas the flow leaving ABC cannot exceed $\mathrm{ub}(x_\mathrm{d}) + \mathrm{ub}(x_\mathrm{e}) = 3 + 1 = 4$, obviously a contradiction. Referring to Example 4.3 again, the generated explanation is the sum of (A), (B) and (C), also showing the contradiction:

$$\text{(ABC)} \quad x_\mathrm{d}\{0..3\} + x_\mathrm{e}\{0..1\} - x_\mathrm{f}\{5..10\} \geq 0 \qquad \max \text{LHS} = 3 + 1 - 5 = -1. \quad \square$$

4.4.2 Propagator for circulations – pruning

Pruning is heuristic (incomplete), except in the restricted cases (i) the SCC-based method is used, as we will explain in Section 4.4.3, and the network is binary, that is all arc flows have bounds 0..1, or (ii) the explicit-testing method is used, as we will explain in Section 4.4.4, and all arcs are tested. In those particular cases the flows are pruned to bounds($\mathcal{Z}$) consistency. We describe the pruning heuristics in detail in the next sections.

4.4.3 SCC-based pruning heuristic

If the checking phase is successful we have a feasible circulation f for the current G, l, u. This yields a residual graph as in Definition 4.4, noting that the residual graph depends on the current resident solution f. As noted in the discussion of Algorithm 4.1, any cycle in the residual graph gives an augmenting cycle and indicates that the current resident flow values (i.e. the current support) can change while maintaining feasibility.

Conversely, if an arc in the residual graph does not form part of any cycle, its corresponding support cannot change while maintaining feasibility, and therefore the associated CP variable can be fixed to the current resident flow value. We call this a 'pruning' because it removes unsupported values from the domain of the CP variable.

Therefore we wish to identify arcs which do not form part of any cycle, and prune them. As proposed by Régin [90] for the *alldifferent* propagator, we use Tarjan's Strongly Connected Components (SCC) algorithm [108] to identify cycles and non-cycle arcs.

Definition 4.5 An SCC of the residual graph G^*, is a maximal subset of the nodes N, in which every node is reachable from every other node in the SCC via directed arcs in G^*.

Corollary 4.1 *Any arc in an SCC of G^* is part of a cycle in G^*, since a cycle containing the arc can always be completed from its head to its tail via remaining arcs in the SCC.*

Tarjan's algorithm returns a partition assigning every node in the residual graph to an SCC, with trivial (singleton) SCCs permitted. We instrument the algorithm to also return a Directed Acyclic Graph (DAG) based on a depth-first search order, showing the connectivity of SCCs. The DAG consists of SCCs connected by arcs not in SCCs.

The DAG allows us to produce explanations for the SCC-based prunings. Suppose we have a residual arc +a, with associated CP variable x_a, directed from SCC X to Y. The notation '+' indicates that the residual arc represents an increase in x_a. The explanation will be a *linear* constraint stating that this cannot happen, leading to the removal of the residual arc. Therefore the correct explanation is of the form $x_a \leq \ldots$ in this case.

We sum the node balance equations over the nodes in Y and then relax '=' to '$\geq$' in the same way as in Equation (4.3). Since node balance equations state $f(Y, \overline{Y}) - f(\overline{Y}, Y) = 0$ i.e. outgoing flows - incoming flows = 0, and arc +a is in $(\overline{Y}, Y)$, we obtain an equation $-x_a + \ldots \geq 0$, equivalently $x_a \leq \ldots$, which is what we need. Using Y is arbitrary, we could

have generated a '$\leq$' explanation from X, but given the choice of X or Y, as Rochart et al. [95] has observed we must prune arcs in the correct order to ensure valid explanations.

Theorem 4.3 *Given a residual arc* +a *directed from SCC X to Y and a residual arc* +b *directed from SCC Y to Z, and given an explanation $-x_{\mathrm{a}} + \ldots \geq 0$ from node balance equations of nodes in Y, the explanation does not prune x_{a} (must prune x_{b} first).*

Proof Given arc +b is in $(Y, \overline{Y})$, the explanation has the form $-x_{\mathrm{a}} + x_{\mathrm{b}} + \ldots \geq 0$, equivalently $x_{\mathrm{a}} \leq x_{\mathrm{b}} + \ldots$, which holds substituting current solution values $f(\mathrm{a})$, $f(\mathrm{b}), \ldots$ for the variables in the constraint. The existence of the residual arc +b implies $f(\mathrm{b}) < u(\mathrm{b})$, hence $f(\mathrm{b}) + 1$ is also a legal substitution for x_{b} in the constraint. After this $f(\mathrm{a}) + 1$ becomes a legal substitution for x_{a}, so $f(\mathrm{a}) + 1$ is supported and can't be removed by propagation. □

To constrain $f(\mathrm{a})$ and hence x_{a}, we must constrain x_{b}, reset $u(\mathrm{b})$ to $f(\mathrm{b})$, and remove residual arc +b first. To show this we check the converse of Theorem 4.3, which also holds, proving that SCC-based pruning works. The proof is similar to that of Theorem 4.2.

Theorem 4.4 *Given a residual arc* +a *directed from SCC X to Y and no residual arcs directed from SCC Y to any other SCC, and given an explanation $-x_{\mathrm{a}} + \ldots \geq 0$ from node balance equations of nodes in Y, the explanation prunes x_{a} to '$\leq$' its current support $f(\mathrm{a})$.*

Proof The explanation is more precisely

$$\sum_{b \in (Y, \overline{Y})} x_b - \sum_{b \in (\overline{Y}, Y)} x_b \geq 0,$$

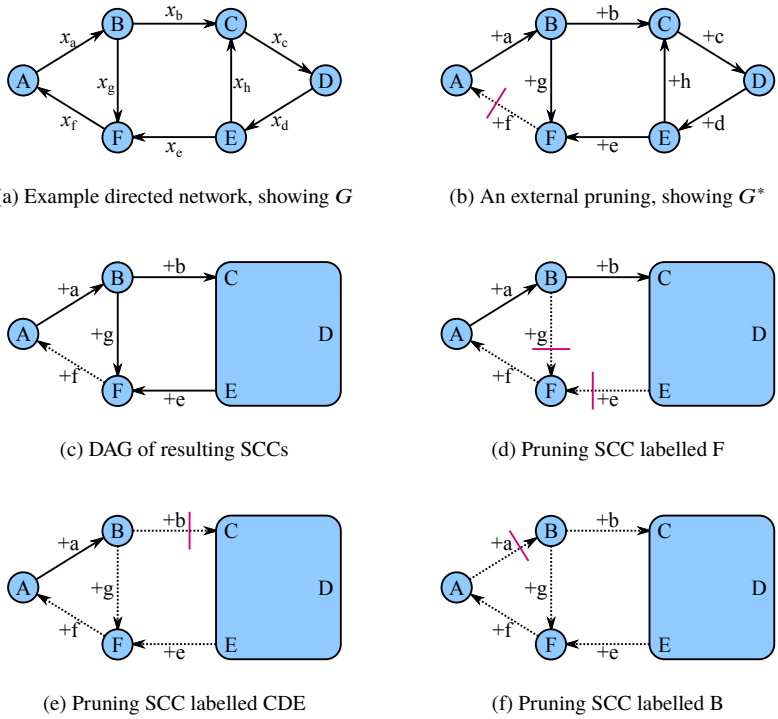
which the *linear* propagator checks by substituting the most generous possible values from $D(x_b)$ so as to maximize the left-hand side, that is, it checks

$$\sum_{b \in (Y, \overline{Y})} \mathrm{ub}(x_b) - \sum_{b \in (\overline{Y}, Y)} \mathrm{lb}(x_b) \geq 0,$$

or simply $u(Y, \overline{Y}) - l(\overline{Y}, Y) \geq 0$. Since no arcs crossing SCC Y gave a residual arc in $(Y, \overline{Y})$, we have $f(b) = u(b)$ for all b in $(Y, \overline{Y})$ and $f(b) = l(b)$ for all b in $(\overline{Y}, Y)$. So the *linear* propagator is equivalently checking $f(Y, \overline{Y}) - f(\overline{Y}, Y) \geq 0$. By flow conservation (4.1) this holds as an equality, hence there is no slack in the constraint. The *linear* propagator sees that substituting $f(x_{\mathrm{a}}) + 1$ into the constraint gives $-1 \geq 0$ or *false* under the most generous possible conditions, hence removes $f(x_{\mathrm{a}}) + 1$ and greater values from $D(x_a)$. □

Considering both theorems we see that SCC-based pruning works provided we take the DAG returned by Tarjan's algorithm, and prune the SCCs in topological order so that whenever an SCC is being pruned, all SCCs reachable from it have already been pruned.

There is always a valid topological ordering of the SCCs, due to the acyclicity of the DAG (which follows from the fact that cycles would cause SCCs to merge). Essentially we



(a) Example directed network, showing G

(b) An external pruning, showing G^*

(c) DAG of resulting SCCs

(d) Pruning SCC labelled F

(e) Pruning SCC labelled CDE

(f) Pruning SCC labelled B

Fig. 4.5 Example execution of the SCC-based pruning algorithm

treat the DAG as a tree and propagate from leaf to root. In our implementation there is no need to consider individual arcs, only SCCs. Each SCC gives a *linear* constraint, which in turn automatically prunes all residual arcs directed towards the given SCC.

Example 4.5 Consider the network of Figure 4.5a representing the node balance equations

$$\begin{array}{lll} \text{(A)} \quad x_a - x_f = 0 & \text{(B)} \quad -x_a + x_b + x_g = 0 & \text{(C)} \quad -x_b + x_c - x_h = 0 \\ \text{(D)} \quad -x_c + x_d = 0 & \text{(E)} \quad -x_d + x_e + x_h = 0 & \text{(F)} \quad -x_e + x_f - x_g = 0. \end{array}$$

All $x_a, x_b, \ldots$ have domain 0..1 with initial supports $f(a), f(b), \ldots = 0$, so that they generate positive residual arcs, i.e. they can increase, giving the residual graph of Figure 4.5b. Figure 4.5b also shows an external pruning where some other propagator sets $x_f = 0$ hence $u(f) = 0$, removing the residual arc +f and changing the SCCs of the residual graph.

Initially there were 3 cycles ABCDEFA, ABFA, CDEC. The removed arc breaks cycles ABCDEFA and ABFA, leaving CDEC as the only cycle and hence the only nontrivial SCC. Tarjan's algorithm starts at node A and performs depth-first search to find the DAG shown

in Figure 4.5c. The explanations (constraints) associated with the 4 SCCs of the graph are

$$\begin{aligned}
\text{(A)} \quad & x_{\mathrm{a}}\{0..1\} - x_{\mathrm{f}}\{0..0\} \geq 0 && \max \text{LHS} = 1 - 0 = 1 \\
\text{(B)} \quad & -x_{\mathrm{a}}\{0..1\} + x_{\mathrm{b}}\{0..1\} + x_{\mathrm{g}}\{0..1\} \geq 0 && \max \text{LHS} = -0 + 1 + 1 = 2 \\
\text{(CDE)} \quad & -x_{\mathrm{b}}\{0..1\} + x_{\mathrm{e}}\{0..1\} \geq 0 && \max \text{LHS} = -0 + 1 = 1 \\
\text{(F)} \quad & -x_{\mathrm{e}}\{0..1\} + x_{\mathrm{f}}\{0..0\} - x_{\mathrm{g}}\{0..1\} \geq 0 && \max \text{LHS} = -0 + 0 - 0 = 0
\end{aligned}$$

Notice that the only constraint which is tight is (F) which has no dependents in the DAG. Propagating this prunes x_{e} and x_{g} as shown in Figure 4.5d, then re-checking shows that

$$\begin{aligned}
\text{(B)} \quad & -x_{\mathrm{a}}\{0..1\} + x_{\mathrm{b}}\{0..1\} + x_{\mathrm{g}}\{0..0\} \geq 0 && \max \text{LHS} = -0 + 1 + 0 = 1 \\
\text{(CDE)} \quad & -x_{\mathrm{b}}\{0..1\} + x_{\mathrm{e}}\{0..0\} \geq 0 && \max \text{LHS} = -0 + 0 = 0.
\end{aligned}$$

(CDE) is now tight and prunes x_{b} as shown in Figure 4.5e, then re-checking again we find

$$\text{(B)} \quad -x_{\mathrm{a}}\{0..1\} + x_{\mathrm{b}}\{0..0\} + x_{\mathrm{g}}\{0..0\} \geq 0 \qquad \max \text{LHS} = -0 + 0 + 0 = 0,$$

so that finally (B) has become tight and prunes x_{a} as shown in Figure 4.5f, which leaves the network consistent and concludes the SCC-based pruning algorithm. □

Following Gent et al. [54], we incrementally partition down a branch of the search tree, only re-evaluating partitions that change, and re-merging partitions upon backtracking.

4.4.4 Explicit-testing pruning heuristic

If the checking phase is successful we have a feasible circulation f for the current G, l, u. Now given an arc $a \in A$, we don't necessarily have support for every value in $D(x_{\mathrm{a}})$, only a general support utilizing some particular value in $D(x_{\mathrm{a}})$ to show the constraint is satisfiable as a whole. In order to prune $D(x_{\mathrm{a}})$ tightly we must find supports for $\mathrm{lb}(x_{\mathrm{a}})$ and $\mathrm{ub}(x_{\mathrm{a}})$, or alternatively if no support exists, we need to find the tightest bounds which are supported. In the latter case, we also have to provide explanations for the new bounds.

Suppose we are checking and/or pruning $\mathrm{ub}(x_{\mathrm{a}})$ as opposed to $\mathrm{lb}(x_{\mathrm{a}})$, the $\mathrm{lb}(x_{\mathrm{a}})$ case is similar. Given the current support $f(\mathrm{a}) \in D(x_{\mathrm{a}})$, we can generate a family of supports for greater values of x_{a} by repeatedly augmenting cycles in the residual graph containing $+a$, as we did in Algorithm 4.1 when $f(\mathrm{a})$ was below $l(\mathrm{a})$ and had to increase.

Algorithm 4.2 shows how to do this, with the changes from Algorithm 4.1 highlighted in bold. The procedure repeats until no further increase to $f(\mathrm{a})$ is possible, and therefore $f(\mathrm{a})$ has reached the new value for $\mathrm{ub}(x_{\mathrm{a}})$. The algorithm then returns a certificate at step 5.

We now show that this certificate yields an explanation for pruning $\mathrm{ub}(x_{\mathrm{a}})$.

Algorithm 4.2 Breadth-first labelling algorithm to test $\mathrm{lb}(x_a)$ or $\mathrm{ub}(x_a)$ given arc a

Step 1. Start with any integral valued f that satisfies the flow conservation conditions (4.1) **and the flow bounds conditions** (4.2).
Step 2. Create a list Q which will be partitioned at position i into nodes already visited and nodes to visit. Initially $Q = []$ and $i = 1$.
Step 3.

(a) **If testing $\mathbf{ub}(x_a)$ and $f(a) < u(a)$**: Set $s \leftarrow \mathcal{H}(a)$, $t \leftarrow \mathcal{T}(a)$. **Label s with $[+a, \epsilon(s) = u(a) - f(a)]$**, and then if $s = t$ then go to step 6, else append s to Q and continue.
(b) **If testing $\mathbf{lb}(x_a)$ and $f(a) > l(a)$**: Set $s \leftarrow \mathcal{T}(a)$, $t \leftarrow \mathcal{H}(a)$. **Label s with $[-a, \epsilon(s) = f(a) - l(a)]$**, and then if $s = t$ then go to step 6, else append s to Q and continue.
(c) **Otherwise, bound is confirmed, return no change.**

Step 4. While $i < |Q|$, set $x \leftarrow Q[i]$, $i \leftarrow i + 1$, and visit x as follows:

(a) For b in $\mathcal{T}^{-1}(x)$ with $f(b) < u(b)$ **where $b \neq a$**: Set $y \leftarrow \mathcal{H}(b)$. If y is unlabelled: Label y with $[+b, \epsilon(y) = \min(\epsilon(x), u(b) - f(b))]$, and then if $y = t$ then go to step 6, else append y to Q and continue.
(b) For b in $\mathcal{H}^{-1}(x)$ with $f(b) > l(b)$ **where $b \neq a$**: Set $y \leftarrow \mathcal{T}(b)$. If y is unlabelled: Label y with $[-b, \epsilon(y) = \min(\epsilon(x), f(b) - l(b))]$, and then if $y = t$ then go to step 6, else append y to Q and continue.

Step 5. No breakthrough. Return Q as a certificate that $\mathrm{ub}(x_a) \leq f(a)$ or $\mathrm{lb}(x_a) \geq f(a)$.
Steps 6, 7. Breakthrough. Construct and augment a path P as in Algorithm 4.1 steps 6, 7, then return to step 2.

Theorem 4.5 *If Algorithm 4.2 fails at step 5 returning an upper-bound certificate as a set of nodes Q, the explanation* (4.3)*, constructed by summing the node balance constraints over the nodes in Q and relaxing '=' to '≥', will prune x_a to '≤' its new upper bound $f(\mathrm{a})$.*

Proof The *linear* propagator checks the explanation (4.3) by substituting the most generous possible values from $D(x_b)$ so as to maximize the left-hand side, that is, it checks

$$\sum_{b \in (Q, \overline{Q})} \mathrm{ub}(x_b) - \sum_{b \in (\overline{Q}, Q)} \mathrm{lb}(x_b) \geq 0,$$

or simply $u(Q, \overline{Q}) - l(\overline{Q}, Q) \geq 0$. Note that $a \in (\overline{Q}, Q)$, so the LHS contains $-l(a)$.

We show that the LHS and therefore the slack in the constraint is $f(a) - l(a)$, that is, if the constraint were tested with $x_\mathrm{a} = f(\mathrm{a})$ instead of the most generous possible value $l(a)$, it would hold as an equality. Thus higher values of x_a would make the LHS < 0.

By construction Q could not be enlarged by following arcs of G^*, except possibly $-\mathrm{a}$. That is $f(b) = u(b)$ for all b in $(Q, \overline{Q})$ and $f(b) = l(b)$ for all b in $(\overline{Q}, Q) \backslash a$. Then

$$u(Q, \overline{Q}) = f(Q, \overline{Q}) = f(\overline{Q}, Q) = l(\overline{Q}, Q) + f(a) - l(a). \qquad \square$$

This kind of pruning is expensive, so we implement Steiger et al.'s method [106] to determine heuristically a subset of $n\%$ of the $2|A|$ arc bounds to test. We maintain a priority queue

with entries labelled $(+a, score)$ for an upper bound, and $(-a, score)$ for a lower bound, for all a in A, with *score* initially 0. Each time we execute, we test the top $n\%$ of queue entries, re-queueing each with *score* adjusted by +5 if it propagated or −2 if not.

4.4.5 Cut-based pruning heuristic

We also try an innovative pruning heuristic which is suggested by the fact that we frequently generate explanations as *linear* constraints. Our heuristic is, any time we generate a constraint from a certificate as described in Sections 4.4.1 through 4.4.4, we add the constraint (which we call a *cut*) to the constraint database and propagate it using a *linear* propagator.

Given that the basic *circulation* propagator does not do any pruning, it only checks the feasibility of the network, this is potentially a powerful technique since it promises to learn about common failures and then avoid them before they occur, by pre-emptively pruning domains. We hoped that cuts would replace SCC-based pruning (which has been successful for 0..1 flows such as the restricted case of *alldifferent*) for the case of general flow networks, with less runtime burden than the already-proposed explicit-testing heuristic.

Propagating the cuts is expected to become expensive when a large number of cuts have been entered. When we add a cut to the database, we mark it as redundant (as for learnt clauses), and we aggressively discard cuts that seem not worthwhile to propagate. To manage the cut-cache, we employ a variation of the clause-activities scheme used in SAT solvers such as *MiniSAT* [40]. We give each cut an activity counter, which we initially set to 3 and then top up to 3 whenever the cut is involved in a conflict analysis. Every 100 conflicts, we decrement the counters of all cuts, and discard any cut whose counter reaches 0.

4.5 Minimum Cost Flows

Minimum Cost Flow problems are similar to circulation problems in that they consist of a graph $[N; A; \mathcal{T}; \mathcal{H}]$, and vector of flow x_a in each arc $a \in A$, subject to flow conservation, and flow bounds $l_a \le x_a \le u_a$ for all $a \in A$. Minimum Cost Flow problems also have a cost c_a per unit of flow in arc $a \in A$, giving the overall cost $z = \sum_{a \in A} c_a x_a$.

By convention we use vector and matrix notation due to overlap with Linear Programming, thus $\mathbf{x} = [x_a]_{a \in A}$, $\mathbf{l} = [l_a]_{a \in A}$, $\mathbf{u} = [u_a]_{a \in A}$ and $\mathbf{c} = [c_a]_{a \in A}$. We express G as a node/arc incidence matrix, defined by columns as $\mathbf{A} = [\mathbf{e}_{\mathcal{H}(a)} - \mathbf{e}_{\mathcal{T}(a)}]_{a \in A}$, where $\mathbf{e}_n$ is the unit column vector with a '1' in the row for node n. For convenience and by convention we allow b_n = demand (if positive) or supply (if negative) for node $n \in N$, giving a right-hand side vector $\mathbf{b} = [b_n]_{n \in N}$, so that flow balance can be expressed as $\mathbf{Ax} = \mathbf{b}$.

Then the Minimum Cost Flow problem is formally stated as the Linear Program (LP)

$$\min \mathbf{cx} \text{ s.t. } \mathbf{Ax} = \mathbf{b}, \quad \mathbf{l} \le \mathbf{x} \le \mathbf{u}. \tag{4.5}$$

This can be solved for $\mathbf{x}$ using Simplex methods which are normally specialized to the case of a network matrix as here, and extremely efficient. Primal or Dual Simplex methods are available; we prefer the dual for reasons that we will explain in Section 4.9.1. In the following we develop a Dual Network Simplex algorithm which is instrumented for producing explanations, and some relevant theorems, based on Chvátal's book [26, Chapter 10].

4.6 Dual Simplex algorithm

The Dual Simplex algorithm for general Linear Programs is well known. We reproduce the version from Chvátal's book [26, Chapter 10], a good starting point because it is 'extended' to handle arbitrary bounds on variables, which are always present in CP solvers; the main communication between CP and LP solvers in our system is by variable bounds.

Simplex algorithms in general proceed by iteratively improving a solution. The solution is always a *basic* solution, which means that if there are n variables and m constraints, then $n - m$ variables will be *nonbasic* (fixed, e.g. to zero), which gives a unique solution for the remaining *basic* (computed) variables of which there are m. See the book for details.

We use the subscripts B and N to denote partitioning into basic and nonbasic, e.g. we say $\mathbf{A} = [\mathbf{A}_B\ \mathbf{A}_N]$ to partition the constraint matrix by columns, up to rearrangement of the columns which we disregard (the *basis header* says how to unscramble the columns of $\mathbf{A}_B$ and $\mathbf{A}_N$ to recover $\mathbf{A}$). We adopt the shorthand $\mathbf{B}$ instead of $\mathbf{A}_B$, as is common.

Algorithm 4.3 shows the Dual Simplex algorithm, from Chvátal [26, box 10.2]. We make the pre- and post-conditions explicit. We correct a minor error, the fact that $\mathbf{w}_B$ was left undefined in step 2 but used in step 5. We clarify that when a nonbasic variable is not at a particular bound, it's at the other bound and the bounds are different. We add an explanatory note regarding the usage of $\mathbf{w}_B$ in step 5 (which anticipates the network version). Note that we use $\overline{\mathbf{c}}$ for the *reduced costs* vector as is conventional; this is simply a notation to distinguish $\overline{\mathbf{c}}$ from the original costs vector $\mathbf{c}$, and the overline does not mean complement.

We now show that Algorithm 4.3 produces infeasibility certificates (or unbounded dual rays), which we use when the algorithm is embedded in an explaining propagator.

Definition 4.6 The Primal Linear Program of Problem (4.5) can be written

$$\min \mathbf{cx} \text{ s.t. } \begin{bmatrix} \mathbf{A} \\ \mathbf{I} \\ \mathbf{I} \end{bmatrix} \mathbf{x} \begin{matrix} = \\ \geq \\ \leq \end{matrix} \begin{bmatrix} \mathbf{b} \\ \mathbf{l} \\ \mathbf{u} \end{bmatrix} \tag{4.6}$$

with corresponding Dual Linear Program, as defined by Chvátal [26, Chapter 9],

$$\max \begin{bmatrix} \pi & \overline{\mathbf{c}}^{\text{pos}} & \overline{\mathbf{c}}^{\text{neg}} \end{bmatrix} \begin{bmatrix} \mathbf{b} \\ \mathbf{l} \\ \mathbf{u} \end{bmatrix} \text{ s.t. } \begin{bmatrix} \pi & \overline{\mathbf{c}}^{\text{pos}} & \overline{\mathbf{c}}^{\text{neg}} \end{bmatrix} \begin{bmatrix} \mathbf{A} \\ \mathbf{I} \\ \mathbf{I} \end{bmatrix} = \mathbf{c},\ \overline{\mathbf{c}}^{\text{pos}} \geq \mathbf{0},\ \overline{\mathbf{c}}^{\text{neg}} \leq \mathbf{0}. \tag{4.7}$$

Algorithm 4.3 Procedure for one iteration of Dual Simplex

Preconditions.

$\mathbf{B}$ = a basis, i.e. a square matrix of full rank comprising a subset of the columns of $\mathbf{A}$

$\overline{\mathbf{c}}_N = \mathbf{c}_N - \pi \mathbf{A}_N$ where $\pi = \mathbf{c}_B \mathbf{B}^{-1}$

$\overline{\mathbf{c}}_B = \mathbf{0}$

$\mathbf{x}^*_N$ = vector of fixed $x^*_i \in \{l_i, u_i\}$ such that $\overline{c}_i \geq 0$ if $l_i = x^*_i < u_i$ and $\overline{c}_i \leq 0$ if $l_i < x^*_i = u_i$

$\mathbf{x}^*_B = \mathbf{B}^{-1}(\mathbf{b} - \mathbf{A}_N \mathbf{x}^*_N)$ = vector computed from $\mathbf{x}^*_N$ such that $\mathbf{A}\mathbf{x}^* = \mathbf{b}$

Step 1. If $\mathbf{l}_B \leq \mathbf{x}^*_B \leq \mathbf{u}_B$ then stop: $\mathbf{x}^*$ is an optimal solution. Otherwise, choose the leaving variable: this may be any basic variable x_i with $x^*_i < l_i$ or $x^*_i > u_i$.

Step 2. Let $\mathbf{w}_B = \mathbf{e}_p$ the unit vector with a '1' in position p and with p such that x_i appears in the pth position of the basis heading. Solve the system $\mathbf{vB} = \mathbf{w}_B$, then compute $\mathbf{w}_N = \mathbf{vA}_N$.

Step 3. If $x^*_i < l_i$, then let J be the set of those nonbasic variables x_j for which $w_j < 0, l_j < x^*_j < u_j$ or $w_j > 0, l_j < x^*_j = u_j$. If $x^*_i > u_i$, then let J be the set of those nonbasic variables x_j for which $w_j > 0, l_j = x^*_j < u_j$ or $w_j < 0, l_j < x^*_j = u_j$. If J is empty then stop: the problem is infeasible. Otherwise, find the x_j in J that minimizes $|\overline{c}_j / w_j|$ and let it be the entering variable.

Step 4. Solve the system $\mathbf{Bd} = \mathbf{A}_j$ where $\mathbf{A}_j$ is the entering column of the constraint matrix.

Step 5. Set $t = (x^*_i - l_i)/w_j$ in case $x^*_i < l_i$ and $t = (x^*_i - u_i)/w_j$ in case $x_i > u_i$. Replace the value x^*_j of x_j by $x^*_j + t$ and replace the values $\mathbf{x}^*_B$ of the basic variables by $\mathbf{x}^*_B - t\mathbf{d}$. Replace the leaving column of $\mathbf{B}$ by the entering column and, in the basis heading, replace the leaving variable by the entering variable. Set $\overline{c}_i = -\overline{c}_j / w_j$ and add $\overline{c}_i w_s$ to each $\overline{c}_s$ with $s \neq i$ *[or more simply, set* $\lambda = -\overline{c}_j / w_j$ *and replace* $\overline{\mathbf{c}}$ *by* $\overline{\mathbf{c}} + \lambda \mathbf{w}$*, which will set* $\overline{c}_i = \lambda$ *and* $\overline{c}_j = 0$*; an alternative is to replace* π *by* $\pi - \lambda \mathbf{v}$ *and then regenerate* $\overline{\mathbf{c}}$ *from* π *on demand, preferred in the network case].*

Postconditions. Maintains the preconditions.

Definition 4.7 To relate $\overline{\mathbf{c}}$ and $\mathbf{w}$, referred to in Algorithm 4.3, to Problem (4.7), we define

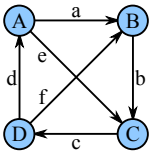
$$(\overline{c}^{\text{pos}}_i, \overline{c}^{\text{neg}}_i) = \begin{cases} (\overline{c}_i, 0), \ \overline{c}_i \geq 0 \\ (0, \overline{c}_i), \ \overline{c}_i < 0 \end{cases} \quad \text{and} \quad (w^{\text{pos}}_i, w^{\text{neg}}_i) = \begin{cases} (w_i, 0), \ w_i \geq 0 \\ (0, w_i), \ w_i < 0. \end{cases}$$

Lemma 4.1 *When Algorithm 4.3 terminates at step 3, the vector* $[-\mathbf{v}\ \mathbf{w}]$ *that was computed at step 2, taken as* $[-\mathbf{v}\ \mathbf{w}^{\text{pos}}\ \mathbf{w}^{\text{neg}}]$ *via Definition 4.7, is an unbounded ray of Problem* (4.7)*.*

Proof We show that if $[\pi\ \overline{\mathbf{c}}^{\text{pos}}\ \overline{\mathbf{c}}^{\text{neg}}]$ is a solution to Problem (4.7) then so is $[\pi\ \overline{\mathbf{c}}^{\text{pos}}\ \overline{\mathbf{c}}^{\text{neg}}] + \lambda[-\mathbf{v}\ \mathbf{w}^{\text{pos}}\ \mathbf{w}^{\text{neg}}]$ for any $\lambda > 0$ and that the latter solution has greater objective. That is,

$$\left[-\mathbf{v}\ \mathbf{w}^{\text{pos}}\ \mathbf{w}^{\text{neg}} \right] \begin{bmatrix} \mathbf{b} \\ \mathbf{l} \\ \mathbf{u} \end{bmatrix} > 0, \ \left[-\mathbf{v}\ \mathbf{w}^{\text{pos}}\ \mathbf{w}^{\text{neg}} \right] \begin{bmatrix} \mathbf{A} \\ \mathbf{I} \\ \mathbf{I} \end{bmatrix} = \mathbf{0}, \ \mathbf{w}^{\text{pos}} \geq \mathbf{0}, \ \mathbf{w}^{\text{neg}} \leq \mathbf{0}. \tag{4.8}$$

$\mathbf{v}$ is row p of $\mathbf{B}^{-1}$ corresponding to x^*_i thus $x^*_i = \mathbf{v}(\mathbf{b} - \mathbf{A}_N \mathbf{x}^*_N)$ from preconditions. Now suppose $x^*_i < l_i$. Then because the algorithm terminated we know that for nonbasic variables, $w_j \geq 0$ when $l_j = x^*_j < u_j$ and $w_j \leq 0$ when $l_j < x^*_j = u_j$, hence $\mathbf{w}_N \mathbf{x}^*_N = w^{\text{pos}}_N \mathbf{l}_N + w^{\text{neg}}_N \mathbf{u}_N$.



$$\mathbf{A} = \left[\begin{array}{c|cccccc} & \mathrm{a} & \mathrm{b} & \mathrm{c} & \mathrm{d} & \mathrm{e} & \mathrm{f} \\ \hline \mathrm{A} & -1 & 0 & 0 & 1 & -1 & 0 \\ \hline \mathrm{B} & 1 & -1 & 0 & 0 & 0 & 1 \\ \mathrm{C} & 0 & 1 & -1 & 0 & 1 & 0 \\ \mathrm{D} & 0 & 0 & 1 & -1 & 0 & -1 \end{array}\right], \quad \mathbf{b} = \left[\begin{array}{c|c} \mathrm{A} & -1 \\ \hline \mathrm{B} & 2 \\ \mathrm{C} & 3 \\ \mathrm{D} & -4 \end{array}\right]$$

Fig. 4.6 Example flow network

Then

$$x_i^* = \mathbf{v}(\mathbf{b} - \mathbf{A}_N \mathbf{x}_N^*) = \mathbf{v}\mathbf{b} - \mathbf{v}\mathbf{A}_N \mathbf{x}_N^* = \mathbf{v}\mathbf{b} - \mathbf{w}_N \mathbf{x}_N^* = \mathbf{v}\mathbf{b} - \mathbf{w}_N^{\mathrm{pos}} \mathbf{l}_N - \mathbf{w}_N^{\mathrm{neg}} \mathbf{u}_N < l_i.$$

Now $\mathbf{w}^{\mathrm{pos}} = [\mathbf{e}_p \ \mathbf{w}_N^{\mathrm{pos}}]$ whereas $\mathbf{w}^{\mathrm{neg}} = [\mathbf{0} \ \mathbf{w}_N^{\mathrm{neg}}]$. Since $\mathbf{e}_p \mathbf{l}_B = l_i$, we substitute and rearrange to show $\mathbf{v}\mathbf{b} - \mathbf{w}^{\mathrm{pos}}\mathbf{l} + \not{l}_i - \mathbf{w}^{\mathrm{neg}}\mathbf{u} < \not{l}_i$ proving the first statement of (4.8). The rest follows from $\mathbf{w} = \mathbf{v}\mathbf{A}$, known from step 2, and from Definition 4.7. The case $x_i^* > u_i$ is similar. □

4.7 Dual Network Simplex algorithm

In this section we discuss the Dual Network Simplex algorithm, which is the specialization of Dual Simplex to networks. This algorithm uses the same spanning-tree data structure as the better-known Primal Network Simplex, but the update on each iteration is modified to reflect step 5 of Algorithm 4.3. This section shows how we can easily read out the certificates we need, such as the unbounded ray $[-\mathbf{v}\ \mathbf{w}]$, from the data structure.

We label the nodes $\mathrm{A}, \mathrm{B}, \ldots$ for m nodes and the arcs $\mathrm{a}, \mathrm{b}, \ldots$ for n arcs. The constraint matrix $\mathbf{A}$ has m rows (nodes) and n columns (arcs). We decompose into row vectors $\mathbf{A}_{\mathrm{A}}, \mathbf{A}_{\mathrm{B}}, \ldots$ or column vectors $\mathbf{A}_{\mathrm{a}}, \mathbf{A}_{\mathrm{b}}, \ldots$. We use a similar notation for costs, demands, and flows, e.g. for the demand vector $\mathbf{b}$ we write scalars $b_{\mathrm{A}}, b_{\mathrm{B}}, \ldots$, and so on.

Then individual node equations are written $\mathbf{A}_i \mathbf{x} = b_i$ where $i \in \{A, B, \ldots\}$. Flow balance also holds for any group of nodes I, that is $(\sum_{i \in I} \mathbf{A}_i)\, \mathbf{x} = \sum_{i \in I} b_i$. This means any group of nodes I may be considered as essentially a single node, by disregarding flows internal to that group of nodes. When summing $\mathbf{A}$-rows this occurs naturally by cancellation.

Theorem 4.6 *Given* $\mathbf{A}$ *and* $\mathbf{b}$ *defining a flow network, where* $b_{\mathrm{A}} + b_{\mathrm{B}} + \ldots = 0$ *(otherwise there is no solution), and the network is connected (otherwise dummy arcs can be added), one of the node equations – we choose node* A*'s – is redundant and may be omitted.*

Proof The incidence vector (list of incident arcs with their orientations) of A is the negation of the group incidence vector of the remaining nodes, that is $\mathbf{A}_{\mathrm{A}} = -\sum_{i \in \{\mathrm{B},\mathrm{C},\ldots\}} \mathbf{A}_i$. Also $b_{\mathrm{A}} = -\sum_{i \in \{\mathrm{B},\mathrm{C},\ldots\}} b_i$. Then the omitted equation is simply the negation of the group flow balance equation for the nodes $\{\mathrm{B}, \mathrm{C}, \ldots\}$, which holds if the individual equations hold. □

Example 4.6 Figure 4.6 shows an example flow network in both graphical and matrix form. Each matrix or vector is partitioned horizontally and vertically – only the lower right panel of each is significant, the rest is for illustrative purposes only, consisting of the row and column labels and the omitted flow balance equation for node A. □

Definition 4.8 Given **A** defining a connected flow network, a *basis network* is any spanning tree of the original network, and a *basic arc* is an arc of the spanning tree. The basis network consists of the original set of m nodes, plus any subset of $m - 1$ arcs which still yields a connected network (equivalently, has no cycles). The *basis matrix* of the basis network, denoted **B**, is the subset of columns of the constraint matrix **A** corresponding to the set of basic arcs, excluding the top line corresponding to the omitted node equation. Thus for an m-node network, **B** is a square matrix of dimension $m - 1$.

We arrange the basis network into a (notional) tree with the distinguished node A (which does not have a node equation) as the root. Then every node except A must be associated with a single arc that connects it to its parent node. (The arc direction is unchanged from the original network, so it may point from parent to child or from child to parent).

Note that, modulo the removal of the top line, **B** also has network structure. The columns may be in any order, hence our explicit labelling; the ordering is called the *basis header*. Because the basis network has no cycle, **B** has full rank (we omit a formal proof).

The Network Simplex procedure is more efficient than general Simplex because the *basis inverse*, denoted $\mathbf{B}^{-1}$, is easily written down by inspecting **B**. Note that the basis **B** is organized as nodes × arcs whereas its inverse $\mathbf{B}^{-1}$ is organized as arcs × nodes.

Lemma 4.2 *A basis matrix* **B***, for a given basis network, has inverse given by*

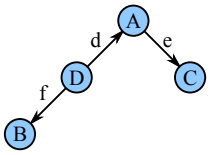
$$[\mathbf{B}^{-1}]_{ji} = \begin{cases} 0, & \text{node } i \text{ is not in the subtree under arc } j \text{ (of the basis network)} \\ 1, & \text{node } i \text{ is in the subtree under arc } j, \text{ and arc } j \text{ points from parent to child} \\ -1, & \text{node } i \text{ is in the subtree under arc } j, \text{ and arc } j \text{ points from child to parent.} \end{cases} \tag{4.9}$$

Then a row (arc) of the basis inverse is simply a list of the nodes appearing in the subtree under that arc, the entire row being negated if the given arc points towards the root.

Equivalently, a column (node) of the basis inverse is a list of the arcs appearing in the path from that node to the root, the signs reflecting the orientations of those arcs.

Proof Given a basic arc j, and I the set of nodes which have a nonzero entry in row j of the basis inverse, and $\mathbf{B}_i$ the incidence vector (list of incident arcs and their orientations) for any node i, we find $\sum_{i \in I} \mathbf{B}_i$ is the incidence vector for the group of nodes I, as we saw earlier when considering group flow balance equations (by cancellation of arcs internal to I).

Referring to Equation (4.9), row j of $\mathbf{B}^{-1}\mathbf{B}$ is $\sum_{i \in I} \mathbf{B}_i$ if arc j points from parent to child or $-\sum_{i \in I} \mathbf{B}_i$ if arc j points from child to parent. The basis network is a tree, so only a single



$$\mathbf{B}^{-1} = \left[\begin{array}{c|ccc} & \text{B} & \text{C} & \text{D} \\ \hline \text{d} & -1 & 0 & -1 \\ \text{e} & 0 & 1 & 0 \\ \text{f} & 1 & 0 & 0 \end{array}\right], \quad \mathbf{B} = \left[\begin{array}{c|ccc} & \text{d} & \text{e} & \text{f} \\ \hline \text{B} & 0 & 0 & 1 \\ \text{C} & 0 & 1 & 0 \\ \text{D} & -1 & 0 & -1 \end{array}\right]$$

Fig. 4.7 A basis for the preceding flow network

arc j is incident to the subtree under j. In the first case $\sum_{i\in I} \mathbf{B}_i = \mathbf{e}_j$; in the second case $\sum_{i\in I} \mathbf{B}_i = -\mathbf{e}_j$. Hence row j of $\mathbf{B}^{-1}\mathbf{B}$ is $\mathbf{e}_j$, the unit vector with a '1' in position j. □

Example 4.7 Figure 4.7 shows a basis network as a spanning tree of the network of Example 4.6. The graphical version is drawn to reflect the root-to-child hierarchy, but this does not affect its structure. We show the basis matrix (omitting root node A) and its inverse.

See that, e.g.: Row d of $\mathbf{B}^{-1}$ reflects the set $I = \{\text{B}, \text{D}\}$ of nodes under arc d, using -1 to indicate inclusion in the set I because d points downwards. Column B of $\mathbf{B}^{-1}$ reflects the set $\{+\text{f}, -\text{d}\}$ of arcs on the path from B to the root with their orientations (+ downwards). □

Corollary 4.2 *The node costs $\pi = \mathbf{c}_B B^{-1}$ are the costs of sending a unit of flow from a given node to the root via the spanning tree, as the sum of the costs of the arcs on that path.*

Theorem 4.7 *When Algorithm 4.3 applies to networks, the vector* $\mathbf{v}$ *computed at step 2 encodes the set of nodes I in the subtree under the leaving arc and the direction of the leaving arc, such that*

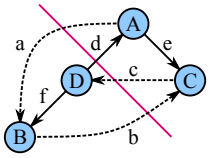
$$v_k = \begin{cases} 0, & \text{node } k \text{ is not in } I \\ 1, & \text{node } k \text{ is in } I \text{ and the leaving arc } i \text{ points from parent to child} \\ -1, & \text{node } k \text{ is in } I \text{ and the leaving arc } i \text{ points from child to parent,} \end{cases}$$

whereas the vector $\mathbf{w}$ *computed at step 2 encodes the set of arcs incident to I, such that*

$$w_k = \begin{cases} 0, & \text{arc } k \text{ is not incident to } I \text{ (it is either internal or external to } I) \\ 1, & \text{arc } k \text{ is incident to } I \text{ with orientation (relative to } I) \text{ the same as leaving arc } i \\ -1, & \text{arc } k \text{ is incident to } I \text{ with orientation (relative to } I) \text{ opposite to leaving arc } i. \end{cases}$$

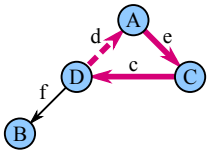
Proof $\mathbf{v}$ is row p of the basis inverse $\mathbf{B}^{-1}$ defined in Lemma 4.2, corresponding to the leaving arc i. $\mathbf{w} = \mathbf{v}\mathbf{A} = \pm \sum_{k\in I} A_k$, i.e. the incidence vector of I ('1' indicates an arc points from not in I to in I), negated if the leaving arc points from child (in I) to parent (not in I). □

Example 4.8 Figure 4.8 shows the same basis network as previously, but nonbasic arcs are shown dotted, and d has been chosen as the leaving arc. The set I of nodes under the leaving arc is $\{\text{B}, \text{D}\}$ and a diagonal line shows the partition of these nodes from the rest.



$$\mathbf{v} = \left[\begin{array}{ccc} \mathrm{B} & \mathrm{C} & \mathrm{D} \\ \hline -1 & 0 & -1 \end{array}\right], \mathbf{A}_N = \left[\begin{array}{c|ccc} & \mathrm{a} & \mathrm{b} & \mathrm{c} \\ \hline \mathrm{B} & 1 & -1 & 0 \\ \mathrm{C} & 0 & 1 & -1 \\ \mathrm{D} & 0 & 0 & 1 \end{array}\right], \mathbf{w}_N = \left[\begin{array}{ccc} \mathrm{a} & \mathrm{b} & \mathrm{c} \\ \hline -1 & 1 & -1 \end{array}\right]$$

Fig. 4.8 Effect of choosing d as the leaving arc



$$\mathbf{A}_\mathrm{c} = \left[\begin{array}{c|c} \mathrm{B} & 0 \\ \mathrm{C} & -1 \\ \mathrm{D} & 1 \end{array}\right], \quad \mathbf{d} = \left[\begin{array}{c|c} \mathrm{d} & -1 \\ \mathrm{e} & 0 \\ \mathrm{f} & 0 \end{array}\right] - \left[\begin{array}{c|c} \mathrm{d} & 0 \\ \mathrm{e} & 1 \\ \mathrm{f} & 0 \end{array}\right] = \left[\begin{array}{c|c} \mathrm{d} & -1 \\ \mathrm{e} & -1 \\ \mathrm{f} & 0 \end{array}\right]$$

Fig. 4.9 Effect of choosing c as the entering arc

Observe that nonbasic arc b crosses the partition in the same orientation as d and has $w_\mathrm{b} = 1$, whereas the other arcs cross in the opposite orientation and have $w_\mathrm{a} = w_\mathrm{c} = -1$. □

Theorem 4.8 *When Algorithm 4.3 applies to networks, the vector* $\mathbf{d}$ *computed at step 4 encodes the unique cycle (as a set of arcs and their orientations) that would be formed if the two ends of the entering arc* j *were joined via the basis network, such that*

$$d_k = \begin{cases} 0, & \text{arc } k \text{ is not in the cycle connecting the ends of } j \text{ via the basis network} \\ -1, & \text{arc } k \text{ is in the cycle with orientation (relative to the cycle) the same as } j \\ 1, & \text{arc } k \text{ is in the cycle with orientation (relative to the cycle) opposite to } j \end{cases}$$

Proof Where arc j points from node u to node v we have $\mathbf{A}_j = \mathbf{e}_v - \mathbf{e}_u$, that is, it has a '1' in the row of node v and a '-1' in the row of node u. Then $\mathbf{d}$ is column v less column u of $\mathbf{B}^{-1}$ as defined in Lemma 4.2. Thus the 'cycle' describes a path from v to the root and then from the root to u (via the basis network), with the common parts of these paths cancelling. □

Example 4.9 Figure 4.9 shows the augmenting cycle in bold arrows, with the leaving arc also dotted to illustrate how the tree will look after completing the iteration. Note that arcs d, e are both clockwise, the same as the entering arc c, so have orientation $d_\mathrm{d} = d_\mathrm{e} = -1$. Since the entering and leaving arcs agree, we should also have $w_\mathrm{c} = -1$ as we see in Figure 4.8. □

With these theorems we can understand the dual pivot, step 5 of Algorithm 4.3, applied to networks. Replacing x_B^* by $x_B^* - t\mathbf{d}$ is augmenting a cycle until the leaving variable x_i reaches the appropriate bound to leave the basis. Replacing π by $\pi - \lambda\mathbf{v}$ is adjusting the costs of sending a unit of flow from nodes in I (the subtree under the leaving arc) to the root, since these flows will now go via the entering arc rather than the leaving arc.

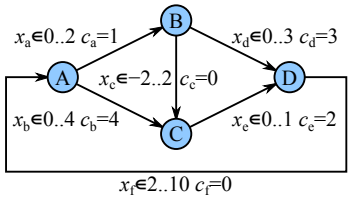


Fig. 4.10 Example of a Minimum Cost Flow problem

4.8 Constraint for Minimum Cost Flows

We define the constraint *min_cost_flow*($\mathbf{A}, \mathbf{b}, \mathbf{c}, \mathbf{x}, z$) for use in CP solvers, as follows

$$\textit{min\_cost\_flow}(\mathbf{A}, \mathbf{b}, \mathbf{c}, \mathbf{x}, z) \equiv (\mathbf{Ax} = \mathbf{b}) \wedge (z \geq \mathbf{cx}). \tag{4.10}$$

As a constraint which either holds or does not hold, this does not of course optimize the objective z by itself, but with this definition the information is available to a branch-and-bound engine (i.e. a CP solver running in optimization mode) to allow z to be assigned to $\mathbf{cx}$ and unpromising searches to be *fathomed*, that is, failed on the basis of cost.

As with the *circulation* constraint, the *min_cost_flow* constraint only enforces the flow conservation and the cost conditions. Thus $\mathbf{l}$ and $\mathbf{u}$ are expressed separately and enforced by the CP solver, and hence taken into account by any *min_cost_flow* propagator.

Example 4.10 The Minimum Cost Flow problem of Figure 4.10, which is the same as Figure 4.3a except that arc costs have been added and are shown beside the arc flow bounds, leads to the Constraint Program (with matrices annotated by node/arc names for clarity)

$$\begin{aligned}
&\text{find } x_a, x_b, x_c, x_d, x_e, x_f, z \in \mathbb{Z} \text{ such that } z \text{ is minimal given} \\
&\quad (0 \leq x_a \leq 2) \wedge (0 \leq x_b \leq 4) \wedge (-2 \leq x_c \leq 2) \\
&\quad \wedge (0 \leq x_d \leq 3) \wedge (0 \leq x_e \leq 1) \wedge (2 \leq x_f \leq 10) \\
&\quad \wedge \textit{min_cost_flow}\left(\left[\begin{array}{c|cccccc} & a & b & c & d & e & f \\ \hline A & -1 & -1 & 0 & 0 & 0 & 1 \\ B & 1 & 0 & -1 & -1 & 0 & 0 \\ C & 0 & 1 & 1 & 0 & -1 & 0 \\ D & 0 & 0 & 0 & 1 & 1 & -1 \end{array}\right], \left[\begin{array}{c|c} A & 0 \\ B & 0 \\ C & 0 \\ D & 0 \end{array}\right], \left[\begin{array}{c|c} a & 1 \\ b & 4 \\ c & 0 \\ d & 3 \\ e & 2 \\ f & 0 \end{array}\right], \left[\begin{array}{c|c} a & x_a \\ b & x_b \\ c & x_c \\ d & x_d \\ e & x_e \\ f & x_f \end{array}\right], z\right).
\end{aligned}$$

When this executes initially the best solution is $\mathbf{x} = [2, 0, 2, 0, 2, 2]$, that is 2 units of flow are sent A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D $\rightarrow$ A. This has cost 6 hence the cost-subconstraint $z \geq \mathbf{cx}$ which is part of the *min_cost_flow* constraint behaves as $z \geq 6$.

Now suppose $\text{ub}(x_c) = 0$. The best solution is then A $\rightarrow$ B $\rightarrow$ D $\rightarrow$ A of cost 8 and in this case the cost constraint behaves as $z \geq 8$. Finally suppose $\text{ub}(x_a) = 0$ also. Then the only solution is A $\rightarrow$ C $\rightarrow$ D $\rightarrow$ A hence the cost constraint behaves as $z \geq 12$. □

4.9 Propagator for Minimum Cost Flows

Similarly to *circulation*, the *min_cost_flow* propagation algorithm consists of distinct phases:

(i) Checking: Does the constraint have any solution to $\mathbf{x}$ given the current bounds $l_a = \text{lb}(x_a)$ and $u_a = \text{ub}(x_a)$ for a in A? If so, the propagator proceeds to the propagation step. If not, the propagator informs the propagation engine to backtrack.

(ii) Pruning based on cost slack: Can the lower bound of z be increased to reflect z^* the cost of the best solution found (since any lesser z would be unsupported by any value of $\mathbf{x}$)? If so prune $\text{lb}(z)$ and fathom if $\text{lb}(z)$ meets $\text{ub}(z)$. Further, for each arc a in A how much could x_a deviate from x_a^*, its optimal value, before $\text{lb}(z)$ meets $\text{ub}(z)$ causing failure (heuristically determined based on the *reduced cost* vector $\overline{\mathbf{c}}$ of the optimal solution)? If this occurs before x_a meets l_a or u_a then prune l_a or u_a as appropriate.

(iii) Pruning based on feasibility: Can some bounds be pruned? That is, for some a in A could we prune $\text{lb}(x_a)$ or $\text{ub}(x_a)$ without cutting off any of the current family of solutions to $\mathbf{x}$? (This is heuristically determined and may also use cost slack).

In any of these cases, the propagator must justify its actions (claiming infeasibility or equivalently fathoming, or pruning domains) by means of explanations. For the purpose of this discussion, an explanation is a globally true redundant *linear* constraint which is a consequence of the *min_cost_flow*. As for *circulation*, clauses can then be extracted.

We now examine the steps and the associated explanations in more detail.

4.9.1 Propagator for Minimum Cost Flows – checking

The propagator sets up $\mathbf{l} = [\text{lb}(x_a)]_{a \in A}$, $\mathbf{u} = [\text{ub}(x_a)]_{a \in A}$ based on the current bounds obtained from the CP solver (for the current subproblem in the search), and then solves for $\mathbf{x}$ using Algorithm 4.3, in its specialized form for networks developed in Section 4.7.

We warm-start the Dual Simplex algorithm using the solution if any from a previous execution from the same propagator. Note that $\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$ might not hold under the new bounds, in which case primal feasibility is lost and a Primal Simplex algorithm would not be warm startable, since it relies on iterative improvement of a resident feasible solution. However *dual* feasibility is not lost by a change in bounds, hence our choice of Dual Simplex which is warm startable, leading to an efficient incremental propagator.

If $\mathbf{x}$ does not exist then Algorithm 4.3 terminates at step 3 yielding a vector $[-\mathbf{v}\ \mathbf{w}]$ which is an *unbounded ray* of an LP which is dual to problem (4.10). The $\mathbf{v}$ component encodes a violating set similar to the Q we saw in Section 4.4.1, which we use for explanations.

The explanation is a redundant constraint taking the form $(\mathbf{vA})\mathbf{x} = \mathbf{vb}$, that is, a weighted sum of the flow balance constraints. There is obviously no correctness issue, any such weighted sum is a valid consequence of *min_cost_flow*. We show that the explanation constructed by means of $\mathbf{v}$ in this way is indeed infeasible, that is, it serves the purpose of cutting off the current search. Note that the explanation can be written $-\mathbf{vb} + \mathbf{wx} = 0$.

Theorem 4.9 *When Algorithm 4.3 yields an unbounded ray* $[-\mathbf{v}\ \mathbf{w}]$*, the redundant constraint* $-\mathbf{vb} + \mathbf{wx} = 0$ *is unsatisfiable under the current bounds condition* $\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$.

Proof We show $-\mathbf{vb} + \mathbf{wx} > 0$. The minimal $\mathbf{wx}$ has $x_i = l_i$ when $w_i \geq 0$ and $x_i = u_i$ when $w_i < 0$. So it's sufficient to show $-\mathbf{vb} + \mathbf{w}^{\text{pos}}\mathbf{l} + \mathbf{w}^{\text{neg}}\mathbf{u} > 0$. We showed this in Lemma 4.1. □

We show that the certificate $[-\mathbf{v}\ \mathbf{w}]$ produced by Algorithm 4.3 on failure encodes a cut-set Q which produces the same explanation (4.3) as produced by the *circulation* propagator. For simplicity we assume $\mathbf{b} = \mathbf{0}$, since *circulation* does not have demands.

Theorem 4.10 *When Algorithm 4.3 yields an unbounded ray* $[-\mathbf{v}\ \mathbf{w}]$*, the unbounded ray encodes the set of nodes I under the leaving arc i chosen at step 1, and consequently a cut-set* Q *which is either* I *or* $\overline{I}$*. Then provided* $\mathbf{b} = \mathbf{0}$*, the explanations*

$$\sum_{a \in (Q, \overline{Q})} x_a - \sum_{a \in (\overline{Q}, Q)} x_a \geq 0 \quad \textit{and} \quad -\mathbf{vb} + \mathbf{wx} \geq 0$$

produced by the circulation and min_cost_flow propagators respectively, are identical.

Proof Referring to Theorem 4.7, we can choose $Q = I$ or $Q = \overline{I}$ so that arcs $a \in (Q, \overline{Q})$ have $w_a = 1$ and arcs $a \in (\overline{Q}, Q)$ have $w_a = -1$. We can ignore the $-\mathbf{vb}$ term since $\mathbf{b} = \mathbf{0}$. □

4.9.2 Propagator for Minimum Cost Flows – pruning based on cost slack

Given the network LP is solveable and an optimal $\mathbf{x}^*$ has been obtained, the next step is to set z greater than or equal to the optimal objective $z^* = \mathbf{cx}^*$. We do this in an indirect manner which (i) provides an explanation for the change to $\text{lb}(z)$, and (ii) prunes, also with an explanation, any x_a, for $a \in A$, which cannot attain its bound on the basis of cost.

When Algorithm 4.3 succeeds and terminates at step 1, it yields a vector π which we use to explain the prunings. Both kinds of pruning are related, being consequences of the redundant constraint $(\mathbf{c} - \pi\mathbf{A})\mathbf{x} \leq z - \pi\mathbf{b}$, that is, the cost constraint minus a weighted sum of the flow balance constraints. Again there is no correctness issue, but we show that the explanation constructed in this manner correctly updates $\text{lb}(z)$ and/or $\text{lb}(x_a)$, $\text{ub}(x_a)$, $a \in A$. Writing the explanation as $\pi\mathbf{b} + \overline{\mathbf{c}}\mathbf{x} \leq z$ shows the connection with reduced costs.

Referring to Definition 4.7, we can split the reduced costs vector $\overline{\mathbf{c}}$ into its positive and negative components, $\overline{\mathbf{c}}^{\text{pos}}$ and $\overline{\mathbf{c}}^{\text{neg}}$, such that $\overline{\mathbf{c}} = \overline{\mathbf{c}}^{\text{pos}} + \overline{\mathbf{c}}^{\text{neg}}$. This yields a complementary primal solution $\mathbf{x}^*$ to the primal problem (4.6), and dual solution $[\pi\ \overline{\mathbf{c}}^{\text{pos}}\ \overline{\mathbf{c}}^{\text{neg}}]$ to the dual

problem (4.7), both of which re-state the Minimum Cost Flow problem (4.10). Implicit in the use of Dual Simplex is that the primal and dual objectives meet at optimality, thus $z^* = \mathbf{c}\mathbf{x}^* = \pi\mathbf{b} + \overline{\mathbf{c}}^{\text{pos}}\mathbf{l} + \overline{\mathbf{c}}^{\text{neg}}\mathbf{u}$ [26, Chapter 10, Strong Duality theorem].

Theorem 4.11 *When Algorithm 4.3 yields a dual solution* $[\pi\ \overline{\mathbf{c}}^{\text{pos}}\ \overline{\mathbf{c}}^{\text{neg}}]$ *of objective* $z^* = \pi\mathbf{b} + \overline{\mathbf{c}}^{pos}\mathbf{l} + \overline{\mathbf{c}}^{neg}\mathbf{u}$*, the redundant constraint* $\pi\mathbf{b} + \overline{\mathbf{c}}\mathbf{x} \leq z$ *implies* $z \geq z*$ *given* $\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$*.*

Proof We show $\pi\mathbf{b} + \overline{\mathbf{c}}\mathbf{x} \geq z^*$. The minimal $\overline{\mathbf{c}}\mathbf{x}$ has $x_i = l_i$ when $\overline{c}_i \geq 0$ and $x_i = u_i$ when $\overline{c}_i < 0$. So it's sufficient to show $\pi\mathbf{b} + \overline{\mathbf{c}}^{\text{pos}}\mathbf{l} + \overline{\mathbf{c}}^{\text{neg}}\mathbf{u} \geq z^*$. But both sides are equivalent. □

Theorem 4.12 *When Algorithm 4.3 yields a primal solution* $\mathbf{x}^*$ *and reduced costs* $\overline{\mathbf{c}}$*, then given an arc* $a \in A$ *and a cost upper bound* z^U*, the redundant constraint* $\pi\mathbf{b} + \overline{\mathbf{c}}\mathbf{x} \leq z$ *implies* $x_a \leq x_a^* + (z^U - z^*)/\overline{c}_a$ *if* $\overline{c}_a$ *is positive, or* $x_a \geq x_a^* + (z^U - z^*)/\overline{c}_a$ *if* $\overline{c}_a$ *is negative.*

Proof To obtain a bound on $\overline{c}_a x_a$ from $\pi\mathbf{b} + \overline{\mathbf{c}}\mathbf{x} \leq z$ we rearrange to

$$\overline{c}_a x_a \leq z - \pi\mathbf{b} - \sum_{b \in A \setminus a} \overline{c}_b x_b. \tag{4.11}$$

We now show that $\overline{c}_a x_a \leq \overline{c}_a x_a^* + z^U - z^*$. Equivalently, since $z^* = \pi\mathbf{b} + \overline{\mathbf{c}}^{\text{pos}}\mathbf{l} + \overline{\mathbf{c}}^{\text{neg}}\mathbf{u}$, we would like to show that $\overline{c}_a x_a \leq \overline{c}_a x_a^* + z^U - \pi\mathbf{b} - \overline{\mathbf{c}}^{\text{pos}}\mathbf{l} - \overline{\mathbf{c}}^{\text{neg}}\mathbf{u}$.

Referring to Algorithm 4.3, $x_a^* = l_a$ when $\overline{c}_a > 0$ or $x_a^* = u_a$ when $\overline{c}_a < 0$. Thus $\overline{c}_a x_a^*$ cancels with one component of either $-\overline{\mathbf{c}}^{\text{pos}}\mathbf{l}$ or $-\overline{\mathbf{c}}^{\text{neg}}\mathbf{u}$, so we have to show that

$$\overline{c}_a x_a \leq z^U - \pi\mathbf{b} - \sum_{b \in A \setminus a} \begin{cases} \overline{c}_b l_b, & \overline{c}_b \geq 0 \\ \overline{c}_b u_b, & \overline{c}_b < 0. \end{cases} \tag{4.12}$$

But (4.12) is simply (4.11) with the most generous possible constants substituted for z and x_b, where $b \in A \setminus a$, hence (4.12) is always implied by (4.11). □

4.9.3 Propagator for Minimum Cost Flows – pruning based on feasibility

The optional additional pruning heuristics for *min_cost_flow* are identical to those already described for *circulation* in Sections 4.4.3 through 4.4.5, with the following minor variations,

1. for the SCC-based method, given a residual arc +a directed from SCC X to Y, the explanation derived from the sum of node balance equations of nodes in Y is $-x_{\text{a}} + \ldots \geq -\sum_{n \in Y} b_n$ instead of $-x_{\text{a}} + \ldots \geq 0$, because *min_cost_flow* has a demand vector $\mathbf{b}$;
2. for the explicit-testing method, due to the requirement of Algorithm 4.3 to maintain a resident dual feasible solution, we do not augment cycles directly as in Algorithm 4.2, but rather, given $a \in A$, to test $\text{ub}(x_a)$ we temporarily set $\text{lb}(x_a)$ to $\text{ub}(x_a)$ and then re-optimize. If $\text{ub}(x_a)$ cannot be attained then a failure explanation $-\mathbf{v}\mathbf{b} + \mathbf{w}\mathbf{x} = 0$ is generated and propagated to set $\text{ub}(x_a)$ to the correct value. Alternatively if $\text{ub}(x_a)$ is attained but z^* exceeds $\text{ub}(z)$ then a cost-slack explanation $\pi\mathbf{b} + \overline{\mathbf{c}}\mathbf{x} \leq z$ suffices.

4.10 Experiments

In these experiments we evaluate the proposed propagators *circulation* and *min_cost_flow*, with and without explanations, on several problems from the literature. We compare against the existing solvers (i) *PaLM* [60], which implements a flow-based *alldifferent* propagator with explanations but without 1UIP conflict analysis, and (ii) *JaCoP* [106], which implements a *min_cost_flow* propagator (called *networkflow*) but without explanations.

We aim to test whether explanations are worthwhile for the flow-based propagators, and whether their use in a modern LCG solver, which has 1UIP conflict analysis, gives an improvement over the best previously existing implementation of each propagator. We further aim to compare the different pruning heuristics (i) no pruning, except the cost-slack based pruning which is always enabled for *min_cost_flow* in all solvers, (ii) *SCC*-based pruning, (iii) explicit testing-based pruning, and (iv) cut-based pruning.

We run on Intel dual processor × quad core Xeon E5405 nodes at 2.0 GHz with 16 Gb RAM per node. All solver executables are single-threaded and each core runs independently as a virtual single-threaded CPU, but we never schedule two different solvers simultaneously on the same physical node. Timeouts are 600s and each core uses 1.5 Gb RAM.

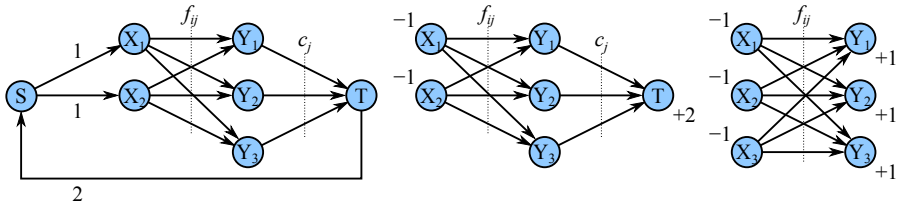
For each problem, we generate 3 sets of 15 instances. The first set consists of easy problems solveable by most of the solvers, the second set are of medium difficulty, and the third set are hard and cause many solvers to time out. (For Social Golfer there are only 2 sets, easy and hard, due to the restricted range of possible instances given the problem structure). We created new instances or modified the publicly available instances as necessary to satisfy these criteria. We have made available on our website `http://people.eng.unimelb.edu.au/pstuckey/flow`, high level models in MiniZinc [80] format.

All solvers use a *first_fail* i.e. smallest-domain-first search strategy, except in Section 4.10.4, where we compare *LCG-glucose* with itself using *LCG-glucose*'s default activity-based search strategy with restarts. For each problem and instance set (easy, medium, or hard), each solver tackles each of the 15 instances 10 times with a different search order, based on randomly permuting the tie-break order of the search strategy in use.

We generate 10 permutations of each FlatZinc [11] model a priori, each having a different tie-break order and hence search. The runtime of a solver on an instance is taken as the median of runtimes over the 10 permutations (with 600 s representing a timeout). For each solver we show the geometric mean of this runtime over the 15 instances in the set.

4.10.1 Encoding *alldifferent* (or *gcc*) to *circulation*

The *alldifferent*($[x_1, \ldots, x_n]$) constraint requires each variable $x_1, \ldots, x_n$ to take a different value. It is used in constructing assignments and permutations, e.g. in rostering or scheduling problems. Taking the domain D of the constraint as $D = D(x_1) \cup \ldots \cup D(x_n)$, each value $v \in D$ occurs 0..1 times. We assume for simplicity that $D = 1..m$ for some $m \geq n$.



(a) For $n = 2$, $m = 3$ (b) Shorthand using demands (c) For $n = 3$, $m = 3$

Fig. 4.11 Example flow networks encoding *alldifferent* to *circulation*

The Global Cardinality Constraint $gcc([x_1, \ldots, x_n], [(v_1, c_1), \ldots, (v_m, c_m)])$ is an extension where the domain of the constraint and the cardinalities of each value are explicitly given, requiring that each value v_i occurs c_i times, for i in $1..m$, with $c_1, \ldots, c_n$ problem variables. A variation is *gcc_low_up* where $c_1, \ldots, c_n$ have an interval domain and don't appear in other constraints (*alldifferent* is a case of *gcc_low_up*). We assume for simplicity that the domain is $D = \{v_1, \ldots, v_m\}$ (that is, *gcc* is *closed*) and that $D = 1..m$.

We encode $alldifferent([x_1, \ldots, x_n])$ to *circulation* as follows. We define nodes S, X_i for i in $1..n$, Y_j for j in $1..m$, T. We define arcs $(\mathrm{X}_i, \mathrm{Y}_j)$ of flow f_{ij} where $f_{ij} = 1$ if $x_i = j$, 0 otherwise. We define arcs $(\mathrm{S}, \mathrm{X}_i)$ of flow 1 to ensure that each x_i takes exactly one value. We define arcs $(\mathrm{Y}_j, \mathrm{T})$ of flow c_j where $c_j = 1$ if value j is taken by any x, 0 otherwise. Finally we define a return arc (T, S) of flow m, i.e. one unit per value taken.

We set $f_{ij} = [x_i = j]$ where $[x_i = j]$ is a Boolean variable which the LCG solver uses internally to track the domain of x_i, taken as an integer 0 = *false* and 1 = *true*. This is very efficient in an LCG solver which already has $[x_i = j]$ literals and integer views thereof. But overall, the decomposition to *circulation* will be slightly less efficient than a dedicated flow-based *alldifferent* propagator, which uses the structure of the network to create explanations based on Hall sets instead of the explicit counting variables $c_j, j \in 1..m$.

Example 4.11 The constraint $alldifferent([x_1, x_2])$ with $x_1, x_2 \in 1..3$ decomposes to the constraint $gcc([x_1, x_2], [(1, c_1), (2, c_2), (3, c_3)])$ with $c_1, c_2, c_3 \in 0..1$. This *gcc* constraint in turn can be encoded as the flow network of Figure 4.11a. □

For simplicity we omit constant parts of the network and simply show the node balance beside each node (as a demand), understanding that this is equivalent to a circulation.

Example 4.12 The network of Figure 4.11a can be drawn more simply as Figure 4.11b. □

When $m = n$, *alldifferent* defines a permutation, and $c_j = 1$, j in $1..n$ can also be omitted.

Example 4.13 $alldifferent([x_1, x_2, x_3])$ with $x_1, x_2, x_3 \in 1..3$ is drawn as Figure 4.11c. □

Encoding *gcc* is identical except that the counting variables $c_i, i \in 1..m$ are exposed to the modeller, so there is less advantage to a dedicated *gcc* propagator over *circulation*.

Propagating *circulation* to bounds($\mathcal{Z}$) consistency is propagating *alldifferent* to domain consistency, or propagating *gcc* to domain consistency on $x_1, \ldots, x_n$ and bounds($\mathcal{Z}$) consistency on $c_1, \ldots, c_m$, as with Régin's original propagation algorithms [90, 91].

4.10.2 Social Golfer

The Social Golfer problem, prob010 in CSPLib [53] is, given a set of N golfers who will play each other over W weeks, and given the number of groups G and the group size S such that $N = GS$, create a roster allocating each golfer to one of G groups of S golfers per week, such that each golfer plays in the same group with each other golfer at most once.

The Social Golfer problem has an integer formulation as follows,

$$\begin{aligned}
&\text{find } x_{g,s,w} \in 1..N \text{ where } g \in 1..G, s \in 1..S, w \in 1..W \\
&\text{such that } \mathit{alldifferent}([x_{g,s,w}]_{g\in 1..G, s\in 1..S}) && \forall w \in 1..W, && (4.13)\\
&\qquad \mathit{alldifferent}([(x_{g,s,w}, x_{g,t,w})]_{g\in 1..G, s\in 1..S, t\in s+1..S, w\in 1..W}), && && (4.14)\\
&\qquad x_{g,s,1} = (g-1)S + s && \forall g \in 1..G, s \in 1..S, && (4.15)\\
&\qquad x_{g,s,w} < x_{g,s+1,w} && \forall g \in 1..G, s \in 1..S-1, w \in 1..W, && (4.16)\\
&\qquad x_{g,1,w} < x_{g+1,1,w} && \forall g \in 1..G-1, w \in 1..W, && (4.17)\\
&\qquad x_{1,1,w} = 1 && \forall w \in 1..W, && (4.18)\\
&\qquad x_{1,2,w} < x_{1,2,w+1} && \forall w \in 1..W-1, && (4.19)\\
&\text{constants } G = \text{number of groups}, S = \text{group size}, W = \text{number of weeks}, N = GS.
\end{aligned}$$

The *alldifferent* constraints (4.13) simply ensure that the roster is a valid roster uniquely assigning each golfer to a group in each week. The single *alldifferent* constraint (4.14) ensures that the pairs of golfers occurring in each group are distinct across weeks.

The remaining constraints (4.15)–(4.19) break symmetries as follows. Players are interchangeable so we fix the first week completely (4.15). Players in a group are interchangeable so we order them numerically (4.16). Groups in a week are interchangeable so we order them lexically, $[x_{g,s,w}]_{s\in 1..S} < [x_{g+1,s,w}]_{s\in 1..S}$, by looking at the first player in each group (4.17). Weeks are interchangeable so we order them lexically, $[x_{g,s,w}]_{g\in 1..G, s\in 1..S} < [x_{g,s,w+1}]_{g\in 1..G, s\in 1..S}$, by looking at the first 2 players in each week (4.18)–(4.19).

In the implementation, we encode a pair of golfers (i, j) where $i, j \in 1..N, i < j$, as an integer $i + (j-1)(j-2)/2$, so that constraint (4.14) is over integers. Hence we define

$$\begin{aligned}
&\text{let } y_{g,i,j,w} \in 1..N(N-1)/2 \text{ where } g \in 1..G, i, j \in 1..S, i < j, w \in 1..W\\
&\text{such that } y_{g,i,j,w} = a[x_{g,i,w}, x_{g,j,w}]\\
&\text{constants } a[i,j] = \begin{cases} i + (j-1)(j-2)/2, & i < j \\ 0, & \text{otherwise} \end{cases} \qquad \forall i, j \in 1..N
\end{aligned}$$

This makes use of the domain-consistent *element2d* constraint available (with explanations) in all solvers in the experiment, to channel domains correctly between x and y.

Example 4.14 Consider $G = 3, S = 3, W = 2$, with solution as follows,

week	group	golfers	pairs	encoded pairs
1	1	1, 2, 3	(1, 2), (1, 3), (2, 3)	1, 2, 3
	2	4, 5, 6	(4, 5), (4, 6), (5, 6)	10, 14, 15
	3	7, 8, 9	(7, 8), (7, 9), (8, 9)	28, 35, 36
2	1	1, 4, 7	(1, 4), (1, 7), (4, 7)	4, 16, 19
	2	2, 5, 8	(2, 5), (2, 8), (5, 8)	8, 23, 26
	3	3, 6, 9	(3, 6), (3, 9), (6, 9)	13, 31, 34.

It is clear that entries in the 'pairs' or 'encoded pairs' column are distinct across weeks. □

The solvers in this experiment are

1. *PaLM* [60], using its flow-based *alldifferent* propagator with explanations for the constraints (4.13), and encoding the constraint (4.14) as a collection of disequalities, because it does not benefit from domain consistency, and also because *PaLM* cannot handle *alldifferent* constraints with many variables or large domains.
2. *LCG-glucose*, by encoding the *alldifferent* constraints (4.13) to *circulation* using the *circulation* propagator with explanations, plus an arc-consistent *alldifferent* propagator for the constraint (4.14) which does not benefit from domain consistency.

PaLM is an experimental fork of *Choco* v2 [87]. We obtained the latest development snapshot of *PaLM* from the *Choco* v2 Subversion repository (no longer online). It contained a number of bugs, which we have had to correct for the experiments. By permission of the authors, we have made the original *PaLM* code and our patches available on our website `http://people.eng.unimelb.edu.au/pstuckey/flow`.

PaLM offers the following modes of operation, which we compare:

(a) Straight *Choco*. Uses the same event dispatcher as *PaLM*. Uses a different search engine (simple backtracking) and propagators (without explanations). We include this as a control comparison, to allow us to see the overhead added by explanations.

(b) Backjumping search engine. The same as straight *Choco*, except it can backjump to the second-highest level present in an explanation after conflict analysis. In this respect it is identical to *LCG-glucose* or a learning SAT solver such as *GRASP* [73].

(c) Retraction search engine (`mac-dbt` algorithm). Experimental, and was useful for open-shop problems at least [60]. We include this mode only because it is *PaLM*'s default mode, and the only mode which offered nogoods in the unmodified code.

(d) With or without learning. The retraction search engine in *PaLM* offers a nogood database and propagation facility. Since we wanted to compare solvers on identical search strate-

gies, we ported this feature over to the backjumping search engine. The resulting backjumping solver with nogoods is comparable to *LCG-glucose*. Without nogoods, explanations are generated solely for the purpose of guiding search, i.e. determining the backjump level (in case b above), or the constraint to retract (in case c above).

LCG-glucose offers the following modes of operation, which we compare:

(a) The SCC algorithm (Section 4.4.3), or explicit-testing of $n\%$ of arcs where $n = 0, 20, 100$ in our tests (Section 4.4.4), or adding cut constraints on failure (Section 4.4.5). Note that cuts are only applicable to the learning solver, because cut activity information collected during conflict analysis is necessary for managing the cut cache.

(b) With or without learning. Unlike in *PaLM*, turning off nogood recording turns off explanations altogether, so *LCG-glucose* without nogoods is comparable to straight *Choco*, indeed with our bug fixes both solvers find the same solution with the same conflict count if both complete. However, *LCG-glucose* uses a Boolean model of the problem even without learning, a slight disadvantage as compared with *Choco*.

We generate all Social Golfer instances with $g, s, w \in 2..8$. We rank instances by difficulty based on the number of possible pairs of golfers. We then eliminate easy instances by making 100 attempts to solve by propagation using *LCG-glucose* with an arc-consistent *alldifferent* propagator [111] and branching but no backtracking (which does not favour *LCG-glucose* or *Choco* since both have the same propagation strength). After this we take the first 15 remaining instances as the easy set, and the next 15 as the hard set.

Table 4.1a shows the results of the experiment. The best pruning algorithm to use is SCC-based, regardless of learning (note that *Choco* and *PaLM* always use SCC-based pruning). With SCCs but without learning, *LCG-glucose* and *Choco* are about the same, with *Choco* slightly faster on the hard set due to its reduced overhead. Adding learning, *LCG-glucose* closes one more instance in each category and is about 10% faster overall.

Turning on *PaLM*'s explanation capability (as opposed to straight *Choco*) results in a much higher overhead. Without learning, comparing *Choco* with *PaLM* (backjumping) shows that *Choco* ran almost 15 times faster and closed 9 instances as opposed to 5, yet the only difference was simple backtracking as opposed to backjumping.

Turning on *PaLM*'s nogood database and propagation capability (backjumping with learning as opposed to backjumping without learning) improved runtimes only imperceptibly. Clearly, the nogoods are not propagating very often. This is because they are *decision* nogoods [36] as opposed to 1UIP nogoods [73]. That is, *PaLM* uses a different method of conflict analysis than *LCG-glucose*. See Section 2.4.1 for a detailed example.

Since the decision nogood does not contain all decisions, it is possible for a decision nogood to propagate during future search of similar subproblems. But this does not occur very often, because the nogood usually contains almost all decisions. To verify this, we checked that *PaLM*'s nogoods are indeed much longer than *LCG-glucose*'s.

		easy		hard	
solver	algorithm	TO	time s	TO	time s
not learning:					
Choco		6	25.34	10	183.39
PaLM	backjumping	10	364.17	15	600.00
	retraction	15	600.00	15	600.00
LCG-glucose	test 0%	14	500.10	15	600.00
	test 20%	6	45.06	11	282.01
	test 100%	6	24.66	11	206.29
	SCC	6	19.81	11	188.70
learning:					
PaLM	backjumping	10	363.65	15	600.00
	retraction	15	600.00	15	600.00
LCG-glucose	test 0%	6	120.12	11	453.14
	test 20%	5	26.33	10	203.06
	test 100%	6	23.05	10	189.61
	SCC	**5**	**17.99**	**10**	**161.23**
	cut (fail)	5	24.39	10	204.37

(a) domain-consistent *alldifferent* via encoding to *circulation*

		easy		hard	
solver	algorithm	TO	time s	TO	time s
not learning:					
Choco		6	39.57	12	361.94
PaLM	backjumping	11	456.44	15	600.00
	retraction	15	600.00	15	600.00
LCG-glucose		6	42.42	12	396.20
learning:					
PaLM	backjumping	11	454.74	15	600.00
	retraction	15	600.00	15	600.00
LCG-glucose		**6**	**29.59**	**11**	**311.85**

(b) arc-consistent *alldifferent* via encoding to disequalities

Table 4.1 Social Golfer experiment

For pruning based on explicit arc testing, increasing the number of arcs pruned gives a dramatic benefit without learning, but a much smaller benefit with learning. This suggests that learning can discover stronger consequences of *circulation* independently.

Pruning with explicit testing or cuts was quite competitive compared with SCCs, taking about 30% and 40% longer than SCCs on the easy problems respectively, but this gap

reduced by about 10% in each case on the hard problems. So although SCCs are the best in this experiment, the other methods might be useful on a different, hard, model.

Table 4.1b shows a control experiment without using *PaLM*'s *alldifferent* propagator or *LCG-glucose*'s *circulation* propagator. Here we use an arc-consistent propagator or a decomposition to disequalities for the constraints (4.13), in addition to (4.14) which always uses this mode. We see similar behaviour between *PaLM* vs. *LCG-glucose* and not learning vs. learning to before, although fewer modes are available to compare.

Comparing Tables 4.1a and 4.1b shows that domain consistency is always beneficial on this problem. This is a useful result given that LCG solvers did not have a domain-consistent *alldifferent* propagator until the matching-based version given in Chapter 3 or the flow-based version given here. Clearly the best solver for this model is *LCG-glucose* with domain consistency, SCCs, and learning. Furthermore, even better results should be possible using *LCG-glucose*'s default activity-based search with restarts instead of *first_fail*.

4.10.3 Personnel Scheduling

The Personnel Scheduling problem is, constructing a cyclic 24-hour schedule for a call centre which must be continuously staffed, but requires different staffing levels at different times of day according to the volume of incoming calls. The 24-hour day is divided into periods, originally 6 periods thus 4 hours per period, and a shift is composed of 2 periods thus 8 hours per shift. The problem was introduced by Bockmayr et al. [19].

In the original 6-period model, any given period contains workers who started in that period, plus workers who started in the period before, and the sum of these must be greater than or equal to the staffing requirement for that period. The objective is to minimize the total number of workers who work a 4-hour period at any time. That is, the count of excess workers above the requirement in a period, is minimized by taking advantage of workers who are also needed in the period immediately before, or immediately after.

We generated a family of instances denoted P_S_N where P = number of periods (originally 6), S = length of a shift in periods (originally 2), and N = the instance identifier 1..15. Each instance consists of a staffing requirements array of length P, each entry being a uniform random number in 0..120 generated using initial random number seed N. To

implement any S other than 2, we had to change the originally proposed model slightly,

$$\text{find } w_1, \ldots, w_P, f_1, \ldots, f_P \in 1..\max_{t=1}^{P} req_t \text{ minimizing } \sum_{t=1}^{P} w_t \tag{4.20}$$

$$\text{such that } w_t \geq req_t \qquad \forall t \in 1..P, \tag{4.21}$$

$$w_t - w_{circ(t-1)} + f_t - f_{circ(t+S)} = 0 \qquad \forall t \in 1..P, \tag{4.22}$$

$$w_t = \sum_{i=1}^{S} f_{circ(t+i)} \qquad \forall t \in 1..P, \tag{4.23}$$

$$\text{constants } P = \text{periods per day}, S = \text{length of shift in periods},$$

$$req_t = \text{workers required in period } t \qquad \forall t \in 1..P,$$

$$\text{functions } circ(t) = ((t-1) \bmod P) + 1,$$

where 'mod' returns $0..P-1$, and $circ(t) : \mathbb{Z} \to 1..P$ gives circular range-reduction.

Here w_t = the number of workers who work in period t, i.e. from time t to $t+1$, and f_t = the number of workers who become free at time t, i.e. just before period t.

The objective (4.20), bounds constraints (4.21), and node balance constraints (4.22) define a *min_cost_flow* constraint and flow graph, with nodes $1..P$, in which node t refers to time t, i.e. the shift change just before period t, with flow w_t in arcs $(t, circ(t+1))$ and flow f_t in arcs $(t, circ(t-S))$. The node balance constraints (4.22) state that the number of workers in period t = the number of workers in period $t-1$, minus any who stopped at time t, plus any who started at time t, i.e. who stopped at time $t+S$ the previous day.

The normalizing constraints (4.23) arise because the flow network admits many augmenting cycles besides the intended cycle for a worker (who travels from node t to node $t+P$ via the w arcs and then skips back to node t via an f arc). For instance, augmenting the cycle $1 \to 2 \to \ldots \to P \to 1$ creates a worker who never signs on or off, which is not allowed. Therefore workers in period t must sign off at times in $t+1..t+S$.

The solvers in this experiment are (i) *JaCoP* 4.4.0, using its *networkflow* constraint, but without learning, and (ii) *LCG-glucose*, using its *min_cost_flow* constraint, with or without learning. The competing solver *JaCoP* has only explicit arc testing which we set to 0, 20 or 100%, whereas *LCG-glucose* has the pruning strategies outlined in Section 4.10.2, plus cuts based on the objective, which apply to *min_cost_flow* but not *circulation*.

Table 4.2 shows the results of this experiment. The easy, medium and hard instance sets are denoted P_S = 24_4, 36_6, 48_8 respectively, for 4 hour shifts with shift changes every 1 hour, 40 minutes, 30 minutes respectively (larger problems than Bockmayr et al.'s).

Considering the results without learning, clearly *JaCoP* relies strongly on explicit arc testing to deliver useful results. *LCG-glucose* is overall slower than *JaCoP*, and gets some benefit from explicit arc testing but not consistently. This is due to our implementation, which runs Dual Network Simplex once per arc tested, whereas *JaCoP* uses a specialized routine.

		24_4		36_6		48_8	
solver	algorithm	TO	time s	TO	time s	TO	time s
not learning:							
JaCoP	test 0%	9	34.46	14	354.98	15	600.00
	test 20%	6	10.62	12	158.67	15	600.00
	test 100%	3	4.30	10	86.25	13	405.33
LCG-glucose	test 0%	13	189.73	14	337.42	15	600.00
	test 20%	7	7.54	15	600.00	15	600.00
	test 100%	8	18.82	15	600.00	15	600.00
	SCC	7	5.98	14	337.21	15	600.00
learning:							
LCG-glucose	test 0%	0	0.37	4	17.56	11	287.55
	test 20%	0	1.34	6	63.49	13	451.74
	test 100%	0	3.17	8	149.56	14	514.14
	SCC	0	0.34	6	24.23	12	328.95
	cut (fail)	0	0.34	5	25.29	12	366.13
	cut (obj)	0	0.23	**4**	**12.20**	**9**	**307.77**
	cut (both)	**0**	**0.22**	4	14.49	12	294.12

Table 4.2 Personnel Scheduling experiment

For *LCG-glucose* the best algorithm is SCC, surprising because SCC only prunes integer flows opportunistically, though it prunes 0..1 flows to bounds($\mathcal{Z}$) consistency.

Adding learning, *LCG-glucose* is clearly far superior to any of the solvers without learning, regardless of pruning algorithm. The best pruning algorithm is cuts based on the objective, surprising because these cuts will never be stronger than the optimally-pruning objective constraint which *min_cost_flow* generates each time it runs. Since the cuts propagate at a higher priority, they must be avoiding a lot of propagator invocations.

We had expected that cuts based on failure (existence of a feasible flow regardless of the objective), would provide useful pruning for integer flows, comparable to SCCs on 0..1 flows (also based on existence of a feasible flow). However, this wasn't the case.

Interestingly, for this problem the next-best pruning algorithm after cuts based on the objective, was no pruning (except for the built-in objective-based pruning). Indeed, with learning explicit testing is always terrible, with runtimes increasing with percentage tested, as opposed to without learning where explicit testing is helpful. This suggests that learning can discover stronger consequences of *min_cost_flow* independently.

We prepared a similar table without *JaCoP*'s *networkflow* constraint or *LCG-glucose*'s *min_cost_flow* constraint, by decomposition to *linear* constraints. We omit this because no solver solved any instances, illustrating that a dedicated flow-based propagator is essential on this problem. The detailed results showed that optimal solutions were found, but could not be proved optimal without *networkflow*/*min_cost_flow*'s fathoming capability.

Overall, adding learning to *min_cost_flow* gave an enormous improvement, while the other algorithms, which are extremely useful without learning, delivered a lesser or negative benefit. We have often seen this latter effect in our studies of LCG propagators.

As well as the improvement to *min_cost_flow* due to learning, even better results are possible using *LCG-glucose*'s default activity-based search with restarts, whereas activity-based search or restarts are not applicable to *JaCoP* which does not have learning.

4.10.4 Fixed Charge Network Flow and Steiner Tree

Fixed Charge Network Flow (FCNF) [84] is, given a directed network in matrix form $\mathbf{Ax} = \mathbf{b}$ and flow bounds $\mathbf{0} \leq \mathbf{x} \leq \mathbf{u}$, find an optimal flow $\mathbf{x} = [x_a]_{a \in A}$ where A is the set of arcs of the network or column labels of $\mathbf{A}$, subject to (i) variable costs based on $\mathbf{v} = [v_a]_{a \in A}$, $\mathbf{v} \geq 0$ yielding a cost $v_a x_a$ per arc $a \in A$; plus (ii) fixed costs based on $\mathbf{f} = [f_a]_{a \in A}$, $\mathbf{f} \geq 0$ yielding a cost f_a per arc $a \in A$ where $x_a > 0$. FCNF can be stated as follows,

$$\text{find } \mathbf{0} \leq \mathbf{x} \leq \mathbf{u}, z \in \mathbb{Z}^{\geq 0}, \mathbf{0} \leq \mathbf{y} \leq \mathbf{1} \text{ minimizing } z + \mathbf{fy}$$

$$\text{such that } \textit{min\_cost\_flow}(\mathbf{A}, \mathbf{b}, \mathbf{v}, \mathbf{x}, z), \tag{4.24}$$

$$y_a = (x_a > 0) \qquad \forall A \in a, \tag{4.25}$$

$$\text{constants } \mathbf{A} = \text{node/arc incidence matrix}, \mathbf{b} = \text{demands},$$

$$\mathbf{u} = \text{capacities}, \mathbf{v} = \text{variable costs}, \mathbf{f} = \text{fixed costs}.$$

Constraint (4.24) enforces flow conservation and handles the variable part of the objective by setting $z = \mathbf{vx}$, with optimal propagation of $\text{lb}(z)$ and consequent fathoming. Constraint (4.25) sets $y_a \in 0..1$ to the truth value of $x_a > 0$ where $0 = \textit{false}$ and $1 = \textit{true}$, a simple side constraint which makes the model more interesting than just *min_cost_flow*.

We can also use *circulation* instead of *min_cost_flow*, with no z variable and using a CP objective of $\mathbf{vx} + \mathbf{fy}$, which gives weaker propagation. Steiner Tree is FCNF with no variable costs i.e. $\mathbf{v} = \mathbf{0}$, and normally uses the *circulation* version of the model. de Uña et al. also solve Steiner Tree using LCG [35], although their work is more complicated and beyond the scope of our current research (we don't claim state-of-the-art results here).

We adapt Gouveia's instances [56] to create 3 sets of 15 FCNF and 15 Steiner Tree instances with N nodes, where $N = 15, 20, 25$ for the easy, medium and hard sets respectively. The N nodes are randomly positioned in a 2D 100×100 field and each node given a demand as a uniform random integer 0..10 (FCNF) or 0..1 (Steiner Tree), where nonzero indicates a sink node. A single source node is then placed in the corner (5 instances), in the centre (5 instances) or in a random location (5 instances). A pair of opposing arcs connects each pair of nodes, with fixed cost = the floored Cartesian distance between the nodes, and variable cost = floored 10% of the distance (FCNF) or zero (Steiner Tree).

		15		20		25	
propagator	algorithm	TO	time s	TO	time s	TO	time s
linear		13	394.71	11	335.79	13	503.92
circulation	test 0%	12	273.90	12	276.88	13	453.57
	test 20%	12	324.19	12	340.94	13	476.96
	test 100%	12	331.91	14	433.20	13	549.52
	SCC	13	267.49	12	277.97	13	416.09
	cut (fail)	12	246.81	12	279.13	12	411.14
min_cost_flow	test 0%	0	4.29	5	52.40	10	187.20
	test 20%	3	21.02	8	162.90	10	339.81
	test 100%	3	43.71	9	239.02	11	498.91
	SCC	**0**	**3.86**	5	52.50	**10**	**182.13**
	cut (fail)	0	4.34	5	62.54	10	200.04
	cut (obj)	0	4.85	**3**	**58.42**	10	200.23
	cut (both)	0	4.88	4	57.92	10	200.55

Table 4.3 Fixed Charge Network Flow experiment

		15		20		25	
propagator	algorithm	TO	time s	TO	time s	TO	time s
linear		0	0.80	2	23.22	10	104.17
circulation	test 0%	**0**	**0.34**	**0**	**7.37**	**9**	**75.24**
	test 20%	0	0.78	1	18.85	9	116.88
	test 100%	0	2.22	4	44.28	10	209.68
	SCC	0	0.42	0	8.91	9	78.52
	cut (fail)	0	0.45	0	8.58	9	79.77
min_cost_flow	test 0%	0	0.43	0	9.19	9	83.68
	test 20%	0	1.71	5	41.80	10	175.84
	test 100%	0	6.71	7	100.67	10	348.69
	SCC	0	0.48	0	9.85	9	86.86
	cut (fail)	0	0.51	0	10.21	9	86.63

Table 4.4 Steiner Tree experiment

In this experiment we compare the different modes and propagators of *LCG-glucose*, using learning and *LCG-glucose*'s default activity-based search strategy with restarts and phase saving. This is the only reasonable approach for these models, the other solvers and strategies make no progress at all. Given that either model can be solved with either *circulation* or *min_cost_flow*, or by decomposition to *linear* constraints, we want to see if the expected propagator is the best on each problem and by what margin.

Tables 4.3 and 4.4 show the results of this experiment. On FCNF the best pruning method to use is not clear-cut, possibly the result for $N = 20$ is due to noise and SCC is the best method, but no pruning (testing 0% of arcs) is also competitive. On Steiner, no pruning is

the best method. SCC is generally competitive, despite these being general integer flows. Explicit testing is always terrible, with runtimes increasing with percentage tested.

The comparison of pruning methods suggests that clause learning gives the cheapest pruning for the case of integer flows (as opposed to 0..1 flows), and that specialized propagation algorithms are less useful when all of *LCG-glucose*'s SAT-like features are enabled. We have often seen this kind of behaviour in our studies of LCG propagators.

Comparing the different propagators we see that on both models the *circulation* propagator was somewhat beneficial over *linear* as it solved one more hard instance in 20% less time. The *min_cost_flow* propagator was dramatically better than the others on FCNF which has a linear objective, solving 2-3 more hard instances in 50-60% less time, and slightly worse than *circulation* on Steiner Tree where *min_cost_flow* uses a null objective.

The non-learning CP solver or the learning CP solver without flow-based constraints is very much improved by our methods on these problems. It would be interesting to add the acyclicity, *reachable*, *articulations*, or lower bounding constraints of de Uña et al. [35].

4.11 Conclusions

The experiments show that in an LCG solver, a generic flow-based propagator can be useful to propagate constraints such as *alldifferent* which are expressible as flow, as compared with a weaker propagator or decomposition not using flow, and that a modern LCG solver with 1UIP nogoods [73] as opposed to decision nogoods [36] is essential.

Still, we advise caution in encoding global constraints to *circulation* or *min_cost_flow* generally, because in other experiments omitted from this thesis for space reasons, we were unable to duplicate previous results[2] in which various constraints such as *gcc* [91], *gsc* [92] and *sliding_sum* [67] were usefully decomposed in this manner. This is because the latest LCG solver, *LCG-glucose*, has very efficient *linear* and other primitive constraints as compared with previous solvers like *Chuffed* [22] or *CPX* [45], making decomposition to primitives, e.g. via MiniZinc's standard library [80], increasingly attractive.

The experiments demonstrate that LCG changes the tradeoffs for propagation. Whereas expensive propagation algorithms, such as the explicit-testing method implemented by *JaCoP* and discussed in Section 4.4.4, can pay for themselves in CP where the reduction in search is considerable, they have more difficulty paying for themselves in LCG. Similar comments apply to the (cheaper) SCC-based method, it was usually at least competitive, but the benefits were often marginal in LCG as compared with traditional CP.

For problems explicitly using flow, the experiments show that with or without learning, a dedicated flow-based propagator is absolutely essential, since decomposition to primitive

[2]Downing, N., Feydy, T., Stuckey, P.J.: Explaining flow-based propagation. In: N. Beldiceanu, N. Jussien, É. Pinson (eds.) *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems: 9th International Conference, CPAIOR 2012, Nantes, France, May 28 – June 1, 2012. Proceedings*, pp. 146–162. Springer Berlin Heidelberg (2012)

constraints is not useful on these problems, and that the dedicated flow-based propagator with learning is usually vastly superior to the same propagator without learning.

Therefore LCG solvers should definitely offer dedicated flow-based constraints with explanation capability. The standard SCC-based propagation method (extended to explain itself) is cheap and helpful on problems whose flow network has the right structure, whereas the new cut-cache (which is only applicable to LCG solvers) is helpful for problems with costs, since it can often avoid running the propagator which is quite expensive.

Chapter 5

Unsatisfiable-Core Optimization in Constraint Programming

5.1 Introduction

Unsatisfiability-based solving has been an area of active research in the SAT community, leading to *unsatisfiable-core guided* MaxSAT solvers which are increasingly effective on suitable problems. These unsatisfiability-based MaxSAT solvers use a learning SAT solver backend, because learning solvers can easily diagnose infeasibilities and return unsatisfiable cores, that is, sets of soft constraints that cannot hold simultaneously.

We extend unsatisfiability-based solving to Constraint Programming (CP) instead of just MaxSAT. We use a Lazy Clause Generation (LCG) solver, which is a CP solver that can utilize learning. LCG solvers can also easily return unsatisfiable cores. Hence it makes sense to extend the unsatisfiability-based MaxSAT algorithms to CP. This will be helpful on e.g. soft-constraint problems where most of the soft constraints hold.

We published an initial version of our work in a technical report[1] which focused on soft-constraint problems in CP using a Pseudo-Boolean objective. A later version in a different technical report[2] gave some initial experiments for a CP problem with a general linear objective, although this was not the main focus of the report. The current work is considerably evolved from this, and now solves any CP problem with a linear objective.

The basic LSSU (Linear Search Sat/Unsat) algorithm for CP or MaxSAT solving consists in generating a series of satisfiable problems which yield better and better solutions to the original problem, until no further improvement is possible. A core-guided variant of

[1]Downing, N., Feydy, T., Stuckey, P.J.: Unsatisfiable cores for constraint programming. *CoRR abs/1305.1690* (2013). URL `http://arxiv.org/abs/1305.1690`

[2]Downing, N., Feydy, T., Stuckey, P.J.: Unsatisfiable cores and lower bounding for constraint programming. *CoRR abs/1508.06096* (2015). URL `http://arxiv.org/abs/1508.06096`

LSSU is MSU4 [72]. By contrast the basic LSUS (Linear Search Unsat/Sat) algorithm consists in generating a series of unsatisfiable problems until a satisfiable one is generated, such that the first solution found is always optimal. Core-guided variants of LSUS include the MSU1/WPM1/WMSU1 family [6, 48, 69], MaxRes [79], OLL [5, 77], and MSU3 [72].

In this chapter we generalize these unsatisfiable-core guided algorithms from MaxSAT to Constraint Programming (CP), in which the problem objective is an arbitrary linear combination of integer problem variables. The generalization of the unsatisfiable-core algorithms from MaxSAT to CP with an integer linear objective, is likely to be most effective when many objective terms are zero or close to zero. We verify our algorithms by comparing core-guided MaxSAT with CP on a number of suitable problems.

5.2 Algorithmic choices in MaxSAT solving

In Chapter 2, Section 2.7 we gave an overall introduction to MaxSAT, including Examples 2.11 (MaxSAT) and 2.12 (Weighted Partial MaxSAT) which we will use as running examples in this chapter. We also gave a brief survey of the LSSU (Linear Search Sat/Unsat) and LSUS (Linear Search Unsat/Sat) distinction [76] and the most important solving algorithms in these categories: MSU1 [48], WPM1 [6], WMSU1 [69], MaxRes [79], OLL [5, 77], MSU3 [71], and MSU4 [72]. We now discuss the implementation choices in these algorithms and propose some alternatives to the original authors' implementations.

5.2.1 Weighted MaxSAT via clause-splitting

To extend the original MSU1 algorithm from unweighted to weighted MaxSAT, Ansótegui et al. and Manquinho et al. simultaneously proposed the WPM1 [6] and WMSU1 [69] algorithms respectively. These algorithms are very similar, and we only consider WPM1 here.

The WPM1 algorithm is, execute as MSU1, except that on each iteration, given an unsatisfiable core $U_1, \ldots, U_n$ of soft clauses with weights $w_1, \ldots, w_n$, observe that the objective must increase by at least $w_{min} = \min_{i=1}^{n} w_i$, and hence construct a *normalized unsatisfiable core* in which all clauses have the same weight w_{min}. Then, complete the current iteration as MSU1, except that the objective lower bound z_{lb} increases by w_{min} instead of 1.

The normalized unsatisfiable core is constructed by replacing each clause $(w_i, U_i), i \in 1..n$ with the pair $\{(w_{min}, U_i), (w_i - w_{min}, U_i)\}$. The former clause, of weight w_{min}, is temporarily returned to the problem database, but also retained in the normalized unsatisfiable core, causing its database version to be processed in the same way as in MSU1. The latter clause, of weight $w_i - w_{min}$, is just returned to the problem database (provided $w_i > w_{min}$) and not further processed. Since the split portion of the clause returned to the clause database will be treated as a hard clause until it appears in a further unsatisfiable core, this means that WPM1 conservatively treats any soft clause as hard until its weight reduces to zero.

5.2.2 Resolution tracing vs. solving under assumptions

The original core-guided `MSU1` MaxSAT algorithm considers soft clauses directly, which requires a special SAT solver capable of producing a *resolution trace*. We discussed SAT resolution proofs in Section 2.2.2, the difference here being that a resolution-tracing SAT solver has to actually generate and save the proof, rather than just treating it as a notional step that illustrates the actions taken before claiming infeasibility.

The resolution trace is a tree showing the derivation of an empty clause, such that the clauses at the leaves of the tree collectively imply unsatisfiability. These clauses constitute an unsatisfiable subset of the original problem clauses and therefore an unsatisfiable core. Note that in the case of partial MaxSAT solving, it is adequate to take only the soft clauses appearing in the resolution trace and discard the hard clauses, since the soft clauses define an unsatisfiable core when taken with respect to *all* hard clauses.

On the other hand, a standard SAT solver can be used, provided it is capable of solving in terms of *assumptions*, as a collection of literals which are set *true* before solving begins. Then the solver proves unsatisfiability under assumptions, by deriving a clause which is empty except for assumption literals i.e. a collection of assumptions which cannot hold simultaneously. Then resolution tracing is irrelevant, since the final clause (conceptually the empty clause) contains the needed information about conflicting soft clauses.

Whereas resolution tracing is convenient for MaxSAT problems with soft clauses simply assumed to be hard, keeping the problem essentially unmodified, we now look at how an arbitrary MaxSAT problem can be transformed for solving under assumptions.

5.2.3 Transforming MaxSAT to SAT optimization

Any soft problem clause $C \equiv x_1 \vee \ldots \vee x_n$ may be *relaxed* by adding a Boolean violation variable v to create the corresponding hard clause $C' \equiv x_1 \vee \ldots \vee x_n \vee v$. Then v makes C' effectively a soft clause, that is, setting $v = \mathit{false}$ in C' forces C to hold, whereas setting v to *true* or leaving v unconstrained means C does not necessarily hold. And then, constructing an objective containing v gives a conventional optimization problem.

Definition 5.1 (SAT-optimization transformation) Given a MaxSAT problem with soft clauses $C = \{C_1, \ldots, C_n\}$ of weights $w_i, \ldots, w_n$ (1 for unweighted MaxSAT), relax the soft clauses $C_1, \ldots, C_n$ with violation variables $v_1, \ldots, v_n$ to create hard clauses $C'_1, \ldots, C'_n$ controlled by $v_1, \ldots, v_n$. Then define $\mathit{objective} = \sum_{i=1}^{n} w_i v_i$ with $v_i = 0$ (*false*) or 1 (*true*).

(a) Infeasibility by resolution tracing (b) Infeasibility in terms of assumptions

Fig. 5.1 Equivalence of MaxSAT vs. SAT-optimization resolution trees

Example 5.1 The MaxSAT problem (2.1), transformed to use violation variables, would be

$$\text{find } x_1, \ldots, x_3, v_1, \ldots, v_6 \in 0..1 \text{ minimizing } \sum_{i=1}^{6} v_i \text{ such that } \bigwedge_{i=1}^{6} C'_i,$$
$$\text{where } C'_1 \equiv \neg x_1 \vee v_1, \qquad C'_2 \equiv \neg x_2 \vee v_2, \qquad C'_3 \equiv \neg x_3 \vee v_3,$$
$$C'_4 \equiv x_1 \vee x_2 \vee v_4, \qquad C'_5 \equiv x_1 \vee x_3 \vee v_5, \qquad C'_6 \equiv x_2 \vee x_3 \vee v_6. \tag{5.1}$$

Any model of the original problem has a corresponding model of the new problem where the contribution of $C_1, \ldots, C_6$ is replaced by the contribution of $v_1, \ldots, v_6$ to the objective. Based on the original list (2.2) we show a subset of the models of the new problem,

x_1	x_2	x_3	v_1	v_2	v_3	v_4	v_5	v_6	z
0	0	0	0	0	0	1	1	1	3
1	0	0	1	0	0	0	0	1	2
0	1	0	0	1	0	0	1	0	2
1	1	0	1	1	0	0	0	0	2

x_1	x_2	x_3	v_1	v_2	v_3	v_4	v_5	v_6	z
0	0	1	0	0	1	1	0	0	2
1	0	1	1	0	1	0	0	0	2
0	1	1	0	1	1	0	0	0	2
1	1	1	1	1	1	0	0	0	3.

(5.2)

There are also redundant models where $v_1, \ldots, v_6, z$ are higher than x_1, x_2, x_3 would require, but these are irrelevant since they are eliminated through optimization. □

To tackle these SAT-optimization problems we define an assumptions set A, initially containing all violation variables, and then solve as a SAT problem under the assumption that the negations of all literals in A hold. The SAT solver then derives a conflict in terms of assumptions. This set of conflicting assumptions is equivalent to a set of conflicting clauses discovered by tackling the original MaxSAT problem using resolution tracing.

Example 5.2 Consider a resolution-tracing SAT solver tackling the original MaxSAT problem (2.1), with soft clauses simply assumed to be hard. The tracing SAT solver follows the resolution steps shown in Figure 5.1a to prove this unsatisfiable. The unsatisfiable core consists of the soft clauses C_1, C_2, C_4 at the leaves of this resolution tree.

By contrast Figure 5.1b shows a regular SAT solver tackling the SAT-optimization version (5.1) of the same problem, with initial assumptions set $A = \{v_1, \ldots, v_n\}$, that is, each objective component assumed to be zero. The regular SAT solver asserts $\neg v_1, \ldots, \neg v_6$ and

then follows the resolution steps shown in Figure 5.1b to derive the clause $\neg v_1 \wedge \neg v_2 \wedge \neg v_4 \rightarrow$ *false*, proving that these literals cannot hold simultaneously.

Observe that the literals v_1, v_2, v_4 discovered by the regular SAT solver are exactly the violation variables controlling the clauses C_1, C_2, C_4 discovered by the tracing SAT solver. The resolution trees are equivalent except that each node of Figure 5.1b is 'tagged' with a subset of $v_1, \ldots, v_6$ showing all clauses referenced by the subtree rooted at the given node. That is, the resolution process itself tracks the set of conflicting clauses. □

In the usual case that the initial assumptions set A turns out to be too strict, assumptions can easily be removed from the assumptions set and the SAT solver restarted. We call this a warm start, since learnt clauses and other information (e.g. the solver's search heuristics) remain undisturbed. By contrast SAT solvers using resolution tracing require a cold start, since if the initial set of clauses turns out to be too strict, clauses must be removed, which entails deleting learnt clauses as well. Warm starting is more efficient.

5.2.4 Weighted MaxSAT via assumption-splitting

The previous section showed at least the first steps of tackling an unweighted MaxSAT problem using an assumptions set A containing an objective as a set of literals. We can generalize this idea to weighted MaxSAT by replacing the set A with a map from literals to their weights. That is, an initial assumptions map $A = \{v_1 \mapsto w_1, \ldots, v_n \mapsto w_n\}$ corresponds to an objective $z = w_1 v_1 + \ldots + w_n v_n$. An advantage of this scheme over maintaining a database of weighted clauses is that the weights in A are trivially editable.

Conceptually the clause-splitting approach of Section 5.2.1 corresponds to defining the assumptions map A to be a multi-map, e.g. replacing the clause (w_1, U_1) with the pair $\{(w_{min}, U_1), (w_1 - w_{min}, U_1)\}$ is equivalent to replacing $v_1 \mapsto w_1$ with $\{v_1 \mapsto w_{min}, v_1 \mapsto w_1 - w_{min}\}$ in the multi-map A. This is an advantage because clause-splitting incurs overhead, whereas assumption-splitting is merely an accounting act which reuses the violation variable v_1, without having to duplicate the hard clause $U_1' \equiv U_1 \vee v_1$ controlled by v_1.

Examining this more carefully, a multi-map isn't really necessary either, because the weighted MaxSAT solving algorithms such as `WPM1` will immediately perform further processing on the clauses (or assumptions) in the normalized unsatisfiable core, so that overall, the entry $v_1 \mapsto w_1$ gets replaced with $v_1 \mapsto w_1 - w_{min}$, and then completely new clauses and/or assumptions are added to the problem database and/or assumptions map according to the algorithm in use. So under a direct translation of `WPM1` to assumptions, the assumptions map will just hold the *remaining weights* of the violation variables.

Algorithm 5.1 Common framework for SAT optimization

inputs:

D_{orig} = initial domains of variables, usually *unset*
C = set of clauses $\{C_1, \ldots, C_n\}$ where C_i = conjunction of literals
objective = $\{(c_1, x_1), \ldots, (c_n, x_n)\}$ with c_i a coefficient, x_i a literal interpreted as 0..1
algorithm ∈ {`LSSU`, `LSUS`, `WPM1`, `MaxRes`, `OLL`, `MSU3`, `MSU4`}
calc_bounds = whether to identify disjoint unsatisfiable cores first

outputs:

optimum_found(S, z) if solution S of weight z found, or *unsatisfiable*

function *SAT_optimization*$(D_{orig}, C, objective, algorithm, calc\_bounds)$

1 $queue \leftarrow []$
2 **if** $algorithm \in \{$`LSSU`, `LSUS`$\}$ **then** $A \leftarrow \emptyset$ **else** $A \leftarrow \{x \mapsto c : (c, x) \in objective\}$
3 $a_{lb} \leftarrow none$
4 $z_{lb} \leftarrow 0$
5 $z_{ub} \leftarrow \infty$
6 $res \leftarrow unsatisfiable$
7 **while** *true* **do**
8 **if** $\neg calc\_bounds$ **then**
9 **for** $(w_{min}, U) \in queue$ **do**
10 **case** *algorithm* **in**
11 `WPM1`: $(C_{new}, objective_{new}) \leftarrow process\_WPM1(w_{min}, U)$
12 `MaxRes`: $(C_{new}, objective_{new}) \leftarrow process\_MaxRes(w_{min}, U)$
13 `OLL`: $(C_{new}, objective_{new}) \leftarrow process\_OLL(w_{min}, U)$
14 $(C, A) \leftarrow (C \cup C_{new}, A \cup \{x \mapsto c : (c, x) \in objective_{new}\})$
15 $queue \leftarrow []$
16 **if** $algorithm \in \{$`LSUS`, `MSU3`$\}$ **then**
17 let $a_{lb} \leftarrow$ new literal
18 $(C, A) \leftarrow (C \cup CNF(\neg a_{lb} \rightarrow \sum_{(c,x) \in objective} cx \leq z_{lb}), A \cup \{a_{lb} \mapsto 1\})$
19 # defer to SAT oracle, passing hard clauses and assumption literals
20 **case** $SAT(D_{orig}, C, \{\neg x : x \mapsto w \in A, w > 0\})$ **in**
21 *satisfiable*(S): # S = model as map from literal to 0..1 where *false* = 0, *true* = 1
22 $z_{ub} \leftarrow \sum_{(c,x) \in objective} cS(x)$
23 $res \leftarrow optimum\_found(S, z_{ub})$
24 **if** $z_{ub} \leq z_{lb}$ **then return** *res*
25 $C \leftarrow C \cup CNF(\sum_{(c,x) \in objective} cx < z_{ub})$
26 $calc\_bounds \leftarrow false$
27 *unsatisfiable*(U): # U = set of assumption literals that cannot hold simultaneously
28 $U \leftarrow \{\neg x : x \in U\}$ # want set of objective literals where one must be nonzero
29 **if** $U = \emptyset$ **then return** *res*
30 $w_{min} \leftarrow \min_{x \in U} A(x)$
31 $z_{lb} \leftarrow z_{lb} + w_{min}$
32 **if** $z_{lb} \geq z_{ub}$ **then return** *res*
33 **if** $algorithm \in \{$`LSUS`, `MSU3`$\}$ **and** $a_{lb} \neq none$ **then** $U \leftarrow U \cup \{a_{lb}\}$
34 **for** $x \in U$ **do** $A(x) \leftarrow A(x) - w_{min}$
35 **if** $algorithm \in \{$`WPM1`, `MaxRes`, `OLL`$\}$ **and** $|U| > 1$ **then**
36 $queue \leftarrow queue + [(w_{min}, U)]$

5.3 Common SAT-optimization framework

There is substantial commonality between the MaxSAT algorithms. Whilst the published descriptions of the algorithms are quite divergent, we can account for many of the differences as implementation choices (SAT encodings of constraints etc), or optimizations (resolution tracing etc), that are orthogonal to the underlying algorithm.

We propose a unified algorithm which brings the implementation choices into conformity where possible, and removes some optimizations, highlighting the underlying similarities between the algorithms. Genuine differences we either encapsulate in a subroutine (the post-core processing which generates the next SAT problem) or we harmlessly extend to all algorithms (objective lower/upper bounds, objective constraints, etc).

Algorithm 5.1 shows our combined algorithm.

The clause set C consists of hard clauses. The hard clauses can be original hard clauses, or can be original soft clauses with an added violation literal. The *objective* is a collection of violation literals and their weights (other literals in C are problem literals).

The assumptions map A holds the remaining weight of each violation variable as described in the previous section. To replicate the publicly described algorithms, the *calc_bounds* flag should be passed as *true* when *algorithm* = `MSU3`, and *false* otherwise.

The algorithm proceeds iteratively. The objective value of an optimal model, if any, lies between z_{lb} and z_{ub}, and this interval tightens at each iteration. An iteration has four main parts, (i) at lines 8–18, applying enqueued problem modifications, (ii) at lines 19–20, solving the current SAT problem, (iii) at lines 21–26, if the problem is satisfiable, decreasing z_{ub} and constraining the problem to approach unsatisfiability, or (iv) at lines 27–37, if the problem is unsatisfiable, increasing z_{lb} and relaxing the problem to approach satisfiability.

Part ii is straightforward and uses the SAT solver outlined in Section 2.2. For satisfiable problems this returns a model as a valuation to the variables of the problem which satisfies the constraints and assumptions. For unsatisfiable problems this returns an unsatisfiable core as a set of assumptions which cannot hold simultaneously, or the empty set if the SAT solver proves the constraint set unsatisfiable regardless of any assumptions.

Part iii applies to the `LSSU`-derived algorithms. Line 22 calculates the objective value of the model found by the SAT solver and sets z_{ub} accordingly, then line 25 posts a new constraint that this must improve in the next model found. For the `LSUS`-derived algorithms, part iii is a no-operation due to the test at line 24, since the first model found will always have an objective equal to z_{lb}, resulting in $z_{lb} = z_{ub}$ and no improvement possible.

Parts i and iv comprise the core of the unsatisfiability-based solving algorithms. Part iv at lines 28–32 is essentially housekeeping, normalizing the unsatisfiable core by calculating w_{min} which is the increase in z_{lb}, applying this increase, and checking for special cases. Lines 33–34 remove all assumptions comprising the normalized unsatisfiable core, from the

Algorithm 5.2 The published `WPM1` algorithm

input: $\varphi = \{(C_1, w_1), \ldots, (C_m, w_m)\}$

```
1  cost ← 0                                          # Optimal
2  while true do
3      (st, φ_c) ← SAT({C_i : (C_i, w_i) ∈ φ})       # Call to the SAT solver without weights
4      if st = SAT then return cost
5      BV ← ∅                                        # Blocking variables of the core
6      w_min ← min{w_i : C_i ∈ φ_c and C_i is soft}
7      for each C_i ∈ φ_c do
8          if C_i is soft then
9              b_i ← new blocking variable
10             φ ← φ\{(C_i, w_i)} ∪ {(C_i, w_i − w_min)} ∪ {(C_i ∨ b_i, w_min)}
11                                                   # Duplicate soft clauses of the core
12             BV ← BV ∪ {b_i}
13     if BV = ∅ then return UNSAT                   # There are no soft clauses in the core
14     else φ ← φ ∪ CNF(∑_{b∈BV} b = 1)               # Add cardinality constraint as hard clauses
15     cost ← cost + w_min
```

assumptions map A. Then part iv lines 35–36 and part i lines 9–15 partially reinstate those assumptions, by calling an appropriate *process_«algorithm»*() backend function.

We discuss the backend functions, and the `MSU3/4` algorithms which don't use a backend function, in a subsequent section for each individual unsatisfiable-core algorithm.

For now, consider `LSSU` (ordinary branch-and-bound optimization), as opposed to `LSUS` (simple increasing-objective optimization). For `LSSU`, optimization happens by decreasing z_{ub} at line 22, whereas all of parts iv and i are a no-operation since the assumptions map is empty. For `LSUS`, optimization happens by generating a special assumption a at lines 16–18, which makes the SAT solver search for a model of objective $\leq z_{lb}$. Since we put a in A with a weight of 1, and the unsatisfiable core must be $\{a\}$ or $\emptyset$, z_{lb} increases by 1 each iteration until the problem becomes satisfiable or the unsatisfiable core is empty.

5.4 `MSU1/WPM1` unsatisfiable-core algorithm

Algorithm 5.2 shows the published `WPM1` algorithm [6] of which `MSU1` [48] is a special case. Given a set of weighted clauses, line 3 solves the corresponding SAT problem treating soft clauses as hard, obtaining an unsatisfiable core φ_c if this fails. Line 6 calculates the minimum weight w_{min} of clauses in φ_c. Line 10 splits each clause C_i in φ_c of weight w_i into two clauses of weights $w_i - w_{min}$ and w_{min}, and relaxes the latter clause with a new blocking variable. Line 14 then sets up the next problem to solve so that only one blocking variable is *true*, i.e. only one of the clauses of weight w_{min} is relaxed in the next problem.

Example 5.3 Consider how `WPM1` tackles the MaxSAT problem (2.1), having clauses

$$C_1 \equiv \neg x_1, \qquad C_2 \equiv \neg x_2, \qquad C_3 \equiv \neg x_3,$$
$$C_4 \equiv x_1 \vee x_2, \qquad C_5 \equiv x_1 \vee x_3, \qquad C_6 \equiv x_2 \vee x_3,$$

each with weight 1. As we saw in Example 5.2, the first SAT call is

$$\varphi = \{(C_1, 1), (C_2, 1), (C_3, 1), (C_4, 1), (C_5, 1), (C_6, 1)\}$$
$$SAT(\{C_1, C_2, C_3, C_4, C_5, C_6\}) = (UNSAT, \{C_1, C_2, C_4\}),$$

with the resolution tree of Figure 5.1a. Splitting these clauses and relaxing the new copies of C_1, C_2, C_4 with new blocking variables b_1, b_2, b_3 gives the next problem,

$$\varphi = \{\cancel{(C_1, 1)}\ \mathbf{0}), (\boldsymbol{C_1 \vee b_1}, \mathbf{1}), \cancel{(C_2, 1}\ \mathbf{0}), (\boldsymbol{C_2 \vee b_2}, \mathbf{1}), (C_3, 1),$$
$$\cancel{(C_4, 1}\ \mathbf{0}), (\boldsymbol{C_4 \vee b_3}, \mathbf{1}), (C_5, 1), (C_6, 1)\} \cup \boldsymbol{HARD(CNF(b_1 + b_2 + b_3 = 1))},$$

where **boldface** denotes additions, ~~strikeout~~ denotes removals or changes, and cancellation denotes clauses which are not stored due to their weight being reduced to 0. We write $HARD(clauses)$ for the hard version $\{(C, \infty) : C \in clauses\}$ of $clauses$.

Now consider the original list (2.2) of possible models for the problem in Example 2.11. In every row there must be a '1' indicating a violation, in the column for C_1, C_2 or C_4, otherwise φ_c would not be an unsatisfiable core. We set one of b_1, b_2 or $b_3 = 1$ according to where this occurs (considering only the first occurrence), to 'hide' a violation from the objective z. This gives a subset of the non-dominated models of the new problem,

x_1	x_2	x_3	$\boldsymbol{b_1}$	$\boldsymbol{b_2}$	$\boldsymbol{b_3}$	$C_1 \vee \boldsymbol{b_1}$	$C_2 \vee \boldsymbol{b_2}$	C_3	$C_4 \vee \boldsymbol{b_3}$	C_5	C_6	z	
0	0	0	**0**	**0**	**1**	0	0	0	+**0**	1	1	~~3~~ **2**	
1	0	0	**1**	**0**	**0**	+**0**	0	0	0	0	1	~~2~~ **1**	
0	1	0	**0**	**1**	**0**	0	+**0**	0	0	1	0	~~2~~ **1**	
1	1	0	**1**	**0**	**0**	+**0**	1	0	0	0	0	~~2~~ **1**	(5.3)
0	0	1	**0**	**0**	**1**	0	0	1	+**0**	0	0	~~2~~ **1**	
1	0	1	**1**	**0**	**0**	+**0**	0	1	0	0	0	~~2~~ **1**	
0	1	1	**0**	**1**	**0**	0	+**0**	1	0	0	0	~~2~~ **1**	
1	1	1	**1**	**0**	**0**	+**0**	1	1	0	0	0	~~3~~ **2**.	

The complete set of non-dominated models is (5.3) plus the following symmetries,

x_1	x_2	x_3	$\boldsymbol{b_1}$	$\boldsymbol{b_2}$	$\boldsymbol{b_3}$	$C_1 \vee \boldsymbol{b_1}$	$C_2 \vee \boldsymbol{b_2}$	C_3	$C_4 \vee \boldsymbol{b_3}$	C_5	C_6	z	
1	1	0	**0**	**1**	**0**	1	+**0**	0	0	0	0	~~2~~ **1**	(5.4)
1	1	1	**0**	**1**	**0**	1	+**0**	1	0	0	0	~~3~~ **2**.	

Algorithm 5.3 Post-core processing for `MSU1/WPM1` in our SAT-optimization

function *process_WPM1*$(w_{min}, \{x_1, \ldots, x_n\})$

1 let $a_1, \ldots, a_n$ = new literals
2 let $[j \leq 1], \ldots, [j \leq n-1]$ = an order-encoding of a new problem variable $j \in 1..n$
3 define convenience literals $[j \leq 0] = 0$ (*false*), $[j \leq n] = 1$ (*true*)
4 **return** $(\bigcup_{i=1}^{n} \{[j \leq i-1] \rightarrow [j \leq i], x_i \wedge [j \leq i-1] \rightarrow a_i, x_i \wedge [j > i] \rightarrow a_i\}$, # C_{new}
5 $\quad\quad \{(w_{min}, a_i) : i \in 1..n\})$ # $objective_{new}$

There are also dominated models in which b_1, b_2, b_3 do not encode a violated clause and relax nothing, leading to an objective z which is higher than it needs to be, e.g.

x_1	x_2	x_3	$\boldsymbol{b_1}$	$\boldsymbol{b_2}$	$\boldsymbol{b_3}$	$C_1 \vee \boldsymbol{b_1}$	$C_2 \vee \boldsymbol{b_2}$	C_3	$C_4 \vee \boldsymbol{b_3}$	C_5	C_6	z
0	0	0	**1**	**0**	**0**	0	0	0	1	1	1	3.

Any model for the original problem gives at least one model for the new problem in which z is reduced by 1. This reduction in z accumulates in the variable *cost*.

For the next iteration, the unsatisfiable core would have to be $\{C_1 \vee b_1, C_2 \vee b_2, C_3, C_5, C_6\}$ to hit all rows of (5.3)–(5.4). Repeating the relaxation process yields $cost = 2$ and a satisfiable problem. That is, the problem generated after 2 iterations has a model with $z = 0$, corresponding to a model of the original problem with $z = 0 + cost = 2$. □

5.4.1 `MSU1/WPM1` algorithm – summary and analysis

Consider the new blocking variables $b_i, \ldots, b_n$ where $n = |\varphi_c|$, created at line 9 of Algorithm 5.2, plus the cardinality constraint $b_1 + \ldots + b_n = 1$ created at line 14. These form a one-hot encoding [17] of what is effectively a new integer problem variable $j \in 1..n$, where $b_i \equiv [j = i]$. In what follows, for simplicity we ignore $b_1, \ldots, b_n$ in favour of j.

5.4.2 `MSU1/WPM1` algorithm – SAT-optimization version

Algorithm 5.3 shows the backend processing for our `MSU1/WPM1` algorithm.

Given an unsatisfiable core $\{\neg x_1, \ldots, \neg x_n\}$, which means that $x_1, \ldots, x_n$ cannot all be zero simultaneously, the backend subroutine receives the set $\{x_1, \ldots, x_n\}$, and a weight w_{min} such that any nonzero $x_i, i \in 1..n$ incurs a cost of w_{min} in the objective.

Note that $x_1, \ldots, x_n$ have been removed from the assumptions map A prior to calling the backend subroutine, or at least had their weight reduced by w_{min}. The backend subroutine has to partially reinstate these assumptions by returning a new objective fragment of the form $w_{min}a_1 + \ldots + w_{min}a_n$ with some appropriate definition of $a_1, \ldots, a_n$.

Line 1 creates a replacement literal a_i for each unsatisfiable-core literal $x_i, i \in 1..n$. Lines 2–3 create a new integer variable $j \in 1..n$ which acts as a selector between $a_1, \ldots, a_n$. Then

lines 4–5 post what is effectively a set of constraints $x_i \wedge [j \neq i] \rightarrow a_i$, so that if some x_i incurs a cost of w_{min}, then so does a_i, *except* in one position defined by j.

Inevitably at least one of $x_1, \ldots, x_n = 1$ (*true*) incurring a cost of w_{min}, otherwise $\{\neg x_1, \ldots, \neg x_n\}$ would not have been an unsatisfiable core. In the next iteration the SAT solver is free to set j such that $x_j = 1$, and set the corresponding $a_j = 0$, thus hiding the cost of x_j from the revised objective. The variable z_{lb} in Algorithm 5.1 tracks the difference between the original and revised objective, similarly to *cost* in Algorithm 5.2.

In the implementation we actually decompose the constraint $x_i \wedge [j \neq i] \rightarrow a_i$ to a pair of constraints $x_i \wedge [j \leq i-1] \rightarrow a_i$ and $x_i \wedge [j > i] \rightarrow a_i$, since an order-encoding [107] is more efficient than the one-hot encoding [17] used by Algorithm 5.2. Thus lines 4–5 return the order-encoding constraints $[j \leq i-1] \rightarrow [j \leq i]$, plus the new constraints defining $a_1, \ldots, a_n$ from $x_1, \ldots, x_n$, plus the new objective terms $w_{min}a_1, \ldots, w_{min}a_n$.

Example 5.4 Consider how Algorithm 5.1 with *algorithm* = `WPM1` tackles the SAT-optimization problem (5.1), which has minimization objective $z = v_1 + \ldots + v_6$ over hard clauses

$$C'_1 \equiv \neg x_1 \vee v_1, \qquad C'_2 \equiv \neg x_2 \vee v_2, \qquad C'_3 \equiv \neg x_3 \vee v_3,$$
$$C'_4 \equiv x_1 \vee x_2 \vee v_4, \qquad C'_5 \equiv x_1 \vee x_3 \vee v_5, \qquad C'_6 \equiv x_2 \vee x_3 \vee v_6.$$

Given D_{orig} which we will take for granted in future examples, the first SAT call is

$$C = \{C'_1, C'_2, C'_3, C'_4, C'_5, C'_6\}$$
$$A = \{v_1 \mapsto 1, v_2 \mapsto 1, v_3 \mapsto 1, v_4 \mapsto 1, v_5 \mapsto 1, v_6 \mapsto 1\}$$
$$SAT(D_{orig}, C, \{\neg v_1, \neg v_2, \neg v_3, \neg v_4, \neg v_5, \neg v_6\}) = \mathit{unsatisfiable}(\{\neg v_1, \neg v_2, \neg v_4\}),$$

with the resolution tree of Figure 5.1b. Then $v_1 + v_2 + v_4 > 0$. Replacing v_1, v_2, v_4 in the assumptions with a_1, a_2, a_3, which are constrained to '$\geq$' the original objective terms except in one position defined by a new integer variable $j \in 1..3$, gives the next problem

$$\begin{aligned} C = \{&C'_1, C'_2, C'_3, C'_4, C'_5, C'_6, \\ &\cancel{\boldsymbol{v_1 \wedge [j \leq 0] \rightarrow a_1}}, \boldsymbol{v_1 \wedge [j > 1] \rightarrow a_1}, \\ &\boldsymbol{v_2 \wedge [j \leq 1] \rightarrow a_2}, \boldsymbol{v_2 \wedge [j > 2] \rightarrow a_2}, \\ &\boldsymbol{v_4 \wedge [j \leq 2] \rightarrow a_3}, \cancel{\boldsymbol{v_4 \wedge [j > 3] \rightarrow a_3}}\} \\ A = \{&v_1 \mapsto \boldsymbol{+0}, v_2 \mapsto \boldsymbol{+0}, v_3 \mapsto 1, v_4 \mapsto \boldsymbol{+0}, v_5 \mapsto 1, v_6 \mapsto 1, \\ &\boldsymbol{a_1 \mapsto 1, a_2 \mapsto 1, a_3 \mapsto 1}\}. \end{aligned}$$

where **boldface** denotes additions, ~~strikeout~~ denotes removals or changes, and cancellation shows clauses which are always satisfied and hence not stored. Similarly to Example 5.3, we

show an 'interesting' subset of the non-dominated models of the revised problem,

$$
\begin{array}{cccccccccccc}
x_1 & x_2 & x_3 & \boldsymbol{j} & \not{+}_1\,\boldsymbol{a_1} & \not{+}_2\,\boldsymbol{a_2} & v_3 & \not{+}_4\,\boldsymbol{a_3} & v_5 & v_6 & z & \\
0 & 0 & 0 & \mathbf{3} & 0 & 0 & 0 & +\mathbf{0} & 1 & 1 & \not{3}\,\mathbf{2} & \\
1 & 0 & 0 & \mathbf{1} & +\mathbf{0} & 0 & 0 & 0 & 0 & 1 & \not{2}\,\mathbf{1} & \\
0 & 1 & 0 & \mathbf{2} & 0 & +\mathbf{0} & 0 & 0 & 1 & 0 & \not{2}\,\mathbf{1} & \\
1 & 1 & 0 & \mathbf{1} & +\mathbf{0} & 1 & 0 & 0 & 0 & 0 & \not{2}\,\mathbf{1} & (5.5) \\
0 & 0 & 1 & \mathbf{3} & 0 & 0 & 1 & +\mathbf{0} & 0 & 0 & \not{2}\,\mathbf{1} & \\
1 & 0 & 1 & \mathbf{1} & +\mathbf{0} & 0 & 1 & 0 & 0 & 0 & \not{2}\,\mathbf{1} & \\
0 & 1 & 1 & \mathbf{2} & 0 & +\mathbf{0} & 1 & 0 & 0 & 0 & \not{2}\,\mathbf{1} & \\
1 & 1 & 1 & \mathbf{1} & +\mathbf{0} & 1 & 1 & 0 & 0 & 0 & \not{3}\,\mathbf{2}. &
\end{array}
$$

As in Example 5.3 the full set of non-dominated models adds the following symmetries,

$$
\begin{array}{cccccccccccc}
x_1 & x_2 & x_3 & \boldsymbol{j} & \not{+}_1\,\boldsymbol{a_1} & \not{+}_2\,\boldsymbol{a_2} & v_3 & \not{+}_4\,\boldsymbol{a_3} & v_5 & v_6 & z & \\
1 & 1 & 0 & \mathbf{2} & 1 & +\mathbf{0} & 0 & 0 & 0 & 0 & \not{2}\,\mathbf{1} & (5.6) \\
1 & 1 & 1 & \mathbf{2} & 1 & +\mathbf{0} & 1 & 0 & 0 & 0 & \not{3}\,\mathbf{2}. &
\end{array}
$$

There are also some dominated models where j does not encode any violation, e.g.

$$
\begin{array}{cccccccccc}
x_1 & x_2 & x_3 & \boldsymbol{j} & \not{+}_1\,\boldsymbol{a_1} & \not{+}_2\,\boldsymbol{a_2} & v_3 & \not{+}_4\,\boldsymbol{a_3} & v_5 & v_6 & z \\
0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 1 & 1 & 1 & 3.
\end{array}
$$

Clearly the non-dominated models (5.5)–(5.6) of the SAT-optimization `WPM1` algorithm after one iteration, are the same as the non-dominated models (5.3)–(5.4) of the published `WPM1` algorithm, up to the different objective encoding (linear terms vs. soft clauses).

The next unsatisfiable core would have to be $\{\neg a_1, \neg a_2, \neg v_3, \neg v_5, \neg v_6\}$ to hit all rows of (5.5)–(5.6). Repeating the relaxation process gives $z_{lb} = 2$ and an optimal model. □

We claim our version of the `WPM1` algorithm using the backend function *process_WPM1*() given in this section, emulates the published `WPM1` algorithm [6].

Theorem 5.1 *Our Algorithm 5.1 with algorithm = `WPM1` on a SAT-optimization problem P, emulates the published Algorithm 5.2 on a MaxSAT problem Q, where P and Q are related by the SAT-optimization transformation given in Definition 5.1.*

Proof (sketch) Construct Algorithm 5.1′ from Algorithm 5.1 by replacing the order-encoding $[j \leq 1], \ldots, [j \leq n-1]$ with a one-hot encoding $c_1, \ldots, c_n$. That is, c_i in Algorithm 5.1′ represents $[j > n-1] \wedge [j \leq n]$ in Algorithm 5.1, with $c_1, \ldots, c_n$ made mutually exclusive by $CNF(c_1 + \ldots + c_n \leq 1)$. Show that Algorithms 5.1′ and 5.1 are equivalent.

Consider the problems P_k, Q_k produced by Algorithms 5.1′ and 5.2 after k iterations. Given that the theorem holds for P_{k-1}, Q_{k-1}, show that if P'_k is defined as P_k generated by Algorithm 5.1′ with the variables $x_1, \ldots, x_n$ passed to *process_WPM1*() simplified out by resolution, then P'_k is related to Q_k by the SAT-optimization transformation. □

Algorithm 5.4 The published `MaxRes` algorithm: *PMRes* function

Input: $\phi = \{C_1, \ldots, C_m\}$
Output: $(I, cost(\phi))$, where I is an optimal assignment

```
1  i ← 0, φ^0 ← φ
2  while true do
3      (issat, κ^i, I) ← SolveSAT(φ^i \ {⋃_{j=1}^{i} (□, 1)})
4      if issat then return (I, i)
5      (φ*, B) ← ReifyCore(φ^i, κ^i)
6      φ^+ ← φ* ∪ {(⋁_{b∈B} b)}
7      φ^{i+1} ← ApplyMaxRes(φ^+, B)
8      i ← i + 1
```

Algorithm 5.5 The published `MaxRes` algorithm: *ReifyCore* function

Input: ϕ, κ
Output: ϕ, B

```
1  B ← ∅
2  for (C_j, 1) ∈ softCls(κ) do
3      φ ← (φ \ {(C_j, 1)}) ∪ {(C_j ↔ b̄_j^i)} ∪ {(b̄_j^i, 1)}
4      B ← B ∪ {b_j^i}
5  return φ, B
```

Algorithm 5.6 The published `MaxRes` algorithm: *ApplyMaxRes* function

Input: $\phi, \{b_1, b_2, \ldots, b_p\}$
Output: ϕ

```
1  for i = 1, ..., p do
2      # Restricted MaxRes[(b_i ∨ ··· ∨ b_p, 1/∞), (b̄_i, 1)]
3      φ ← φ \ {(b_i ∨ bi + 1 ∨ ··· ∨ b_p, 1), (b̄_i, 1)}
4      if i < p then
5          φ ← φ ∪ {(b_{i+1} ∨ ··· ∨ b_p, 1)}
6          φ ← φ ∪ {(d_i ↔ (b_{i+1} ∨ ··· ∨ b_p))} ∪ {(b̄_i ∨ d̄_i, 1)}
7      else
8          φ ← φ ∪ {(□, 1)}
9  return φ
```

5.5 `MaxRes` unsatisfiable-core algorithm

Algorithms 5.4–5.6 show the published `MaxRes` algorithm [79]. Given a set of soft or hard clauses, line 3 of *PMRes*() solves the corresponding SAT problem treating soft clauses as hard, obtaining an unsatisfiable core κ^i if this fails, where i is the iteration count and also the objective lower bound (z_{lb} in Algorithm 5.1 or *cost* in Algorithm 5.2), since this version of `MaxRes` considers partial MaxSAT, not weighted partial MaxSAT.

Line 5 of *PMRes*() calls *ReifyCore*() to locally apply their version of the SAT-optimization transformation of Definition 5.1 to just the soft clauses in the unsatisfiable core, obtaining a

revised clause-set and a set of objective variables B, such that $\neg b_j = C_j$. Once an original soft clause receives the transformation it is invisible to subsequent unsatisfiable cores (see line 2 of *ReifyCore*), but they also post a singleton soft clause for each variable in B, which equals the original soft clause giving an equivalent problem.

Line 7 of *PMRes*() calls *ApplyMaxRes*(), which starts with a long clause $b_1 \vee \ldots \vee b_p$ generated at line 6 of *PMRes*(), and successively resolves this with the singleton soft clauses corresponding to the objective variables in B, using MaxSAT resolution [20]. Each resolution step shortens the long clause by one literal and generates a compensation clause. Eventually the long clause reduces to an empty soft clause *false*, denoted $\square$ in the algorithm, with cost 1. This moves the cost of the unsatisfiable core into a single violated clause.

ApplyMaxRes() uses the resolution rule that a soft or hard clause $b_i \vee \ldots \vee b_p$ resolves with a soft clause $\neg b_i$ to produce a soft resolvent clause $b_{i+1} \vee \ldots \vee b_p$ and a soft compensation 'clause' $\neg b_i \vee \neg(b_{i+1} \vee \ldots \vee b_p)$, where $p = |B|, i \in 1..p$. This is a special case of MaxSAT resolution [20] where one clause is a singleton. *ApplyMaxRes*() then defines $d_i \equiv b_{i+1} \vee \ldots \vee b_p$ to create CNF versions $\neg b_i \vee \neg d_i$ of the compensation clauses.

Example 5.5 Consider how `MaxRes` tackles the MaxSAT problem (2.1), having clauses

$$C_1 \equiv \neg x_1, \qquad C_2 \equiv \neg x_2, \qquad C_3 \equiv \neg x_3,$$
$$C_4 \equiv x_1 \vee x_2, \qquad C_5 \equiv x_1 \vee x_3, \qquad C_6 \equiv x_2 \vee x_3,$$

each with weight 1. The first SAT call is

$$\phi^0 = \{C_1^*, C_2^*, C_3^*, C_4^*, C_5^*, C_6^*\} \text{ where '}*\text{' indicates soft clauses}$$
$$\mathit{SolveSAT}(\{C_1, C_2, C_3, C_4, C_5, C_6\}) = (\mathit{false}, \{C_1, C_2, C_4\}, \mathit{undefined}),$$

with the resolution tree of Figure 5.1a. Thus the unsatisfiable core is $\kappa^0 = \{C_1, C_2, C_4\}$. However, in this example we will pass these to *ReifyCore*() in the order C_4, C_2, C_1, since their MaxSAT resolution process proceeds right to left, and we prefer the opposite.

After *ReifyCore*() and generating the unsatisfiable-core clause $b_1 \vee b_2 \vee b_3$ to seed the MaxSAT resolution, the problem ϕ^+ passed to *ApplyMaxRes*() for simplification is

$$\phi^+ = \{\cancel{C_1^*}\ \boldsymbol{\neg b_3^*}, \cancel{C_2^*}\ \boldsymbol{\neg b_2^*}, C_3^*, \cancel{C_4^*}\ \boldsymbol{\neg b_1^*}, C_5^*, C_6^*, \boldsymbol{b_1 \vee b_2 \vee b_3}\} \cup$$
$$\boldsymbol{CNF(\neg b_3 \leftrightarrow C_1) \cup CNF(\neg b_2 \leftrightarrow C_2) \cup CNF(\neg b_1 \leftrightarrow C_4)},$$

where **boldface** and ~~strikeout~~ show changes relative to the original problem ϕ^0.

The first resolution step is $\boldsymbol{b_1} \vee b_2 \vee b_3$ (hard) resolves with $\boldsymbol{\neg b_1}$ (violation cost 1) to produce the resolvent $b_2 \vee b_3$ (violation cost 1). We can summarize the costs of the original

pair of clauses versus the cost of the resolvent which replaces them, in the table

b_1	$b_2 \vee b_3$	cost of $b_1 \vee b_2 \vee b_3$	cost of $\neg b_1$	cost of $b_2 \vee b_3$
0	0	∞	0	1
1	0	0	1	1
0	1	0	0	0
1	1	0	1	0.

Based on the first row of this table, the resolvent is weaker than the original pair of clauses, but this does not matter since $b_1 \vee b_2 \vee b_3$ was a synthesized redundant constraint, so it still exists even after the redundant version is replaced the resolvent. Thus the first row cannot happen. The last row requires the compensation 'clause' $\neg b_1 \vee \neg(b_2 \vee b_3)$ (violation cost 1). We can summarize this MaxSAT resolution process using the notation

$$\frac{(\boldsymbol{b_1} \vee b_2 \vee b_3, \infty) \quad (\boldsymbol{\neg b_1}, 1)}{(\neg b_1 \vee \neg(b_2 \vee b_3), 1) \quad (b_2 \vee b_3, 1).}$$

The next resolution step is $\boldsymbol{b_2} \vee b_3$ (violation cost 1) resolves with $\boldsymbol{\neg b_2}$ (violation cost 1) to produce b_3 (violation cost 1). Similar reasoning gives us a compensation clause $\neg b_2 \vee \neg b_3$ (violation cost 1). The final resolution step is $\boldsymbol{b_3}$ (violation cost 1) resolves with $\boldsymbol{\neg b_3}$ (violation cost 1) to produce *false* (violation cost 1). No compensation clause is necessary, but we show one to make the construction explicit. The overall summary is then

$$\frac{\dfrac{(\boldsymbol{b_1} \vee b_2 \vee b_3, \infty) \quad (\boldsymbol{\neg b_1}, 1)}{(\neg b_1 \vee \neg(b_2 \vee b_3), 1) \quad (\boldsymbol{b_2} \vee b_3, 1)} \quad \dfrac{(\boldsymbol{\neg b_2}, 1)}{(\neg b_2 \vee \neg b_3, 1) \quad (\boldsymbol{b_3}, 1)} \quad \dfrac{(\boldsymbol{\neg b_3}, 1)}{(\neg b_3 \vee \neg \mathit{false}, 1) \quad (\mathit{false}, 1).}}{}$$

We can encode the above into CNF by defining $d_i = b_{i+1} \vee \ldots \vee b_3$ where $d_3 = \mathit{false}$,

$$\frac{\dfrac{(\boldsymbol{b_1} \vee b_2 \vee b_3, \infty) \quad (\boldsymbol{\neg b_1}, 1)}{(\neg b_1 \vee \neg d_1, 1) \quad (\boldsymbol{b_2} \vee b_3, 1)} \quad \dfrac{(\boldsymbol{\neg b_2}, 1)}{(\neg b_2 \vee \neg d_2, 1) \quad (\boldsymbol{b_3}, 1)} \quad \dfrac{(\boldsymbol{\neg b_3}, 1)}{(\neg b_3 \vee \neg d_3, 1) \quad (\mathit{false}, 1).}}{}$$

Replacing the clauses at the top and right of the above diagram, with those at the bottom and left, gives the problem ϕ^1 tackled by *PMRes* at the next iteration,

soft clauses: $\{\cancel{C_1}\ \cancel{\boldsymbol{\neg b_3 \vee \neg d_3}}, \cancel{C_2}\ \boldsymbol{\neg b_2 \vee \neg d_2}, C_3, \cancel{C_4}\ \boldsymbol{\neg b_1 \vee \neg d_1}, C_5, C_6, \boldsymbol{false}\}$

hard constraints: $\{\boldsymbol{\neg b_3 \leftrightarrow C_1}, \boldsymbol{\neg b_2 \leftrightarrow C_2}, \boldsymbol{\neg b_1 \leftrightarrow C_4}, \cancel{\boldsymbol{d_3 \leftrightarrow false}}, \boldsymbol{d_2 \leftrightarrow b_3}, \boldsymbol{d_1 \leftrightarrow b_2 \vee b_3}\}$.

where **boldface** and ~~strikeout~~ show changes relative to the original problem, and ~~cancellation~~ shows the last compensation clause which is always satisfied and hence not stored (this clause and associated d_3 are optimized out by the '$i = p$' case in *ApplyMaxRes* lines 4–8).

The revised problem ϕ^1, disregarding the soft clause *false*, has the following models,

x_1	x_2	x_3	$\boldsymbol{b_3}$	$\boldsymbol{b_2}$	$\boldsymbol{b_1}$	$\boldsymbol{d_2}$	$\boldsymbol{d_1}$	~~C_1~~ ***true***	~~C_2~~ $\boldsymbol{\neg b_2 \vee \neg d_2}$	C_3	~~C_4~~ $\boldsymbol{\neg b_1 \vee \neg d_1}$	C_5	C_6	z	
0	0	0	**0**	**0**	**1**	**0**	**0**	0	0	0	**+0**	1	1	~~**3**~~ **2**	
1	0	0	**1**	**0**	**0**	**1**	**1**	**+0**	0	0	0	0	1	~~**2**~~ **1**	
0	1	0	**0**	**1**	**0**	**0**	**1**	0	**+0**	0	0	1	0	~~**2**~~ **1**	
1	1	0	**1**	**1**	**0**	**1**	**1**	**+0**	1	0	0	0	0	~~**2**~~ **1**	(5.7)
0	0	1	**0**	**0**	**1**	**0**	**0**	0	0	1	**+0**	0	0	~~**2**~~ **1**	
1	0	1	**1**	**0**	**0**	**1**	**1**	**+0**	0	1	0	0	0	~~**2**~~ **1**	
0	1	1	**0**	**1**	**0**	**0**	**1**	0	**+0**	1	0	0	0	~~**2**~~ **1**	
1	1	1	**1**	**1**	**0**	**1**	**1**	**+0**	1	1	0	0	0	~~**3**~~ **2**.	

Note that this is the same as the list (5.3) of models for the problem generated by `WPM1` after one iteration, up to the slightly different encoding of the added variables b_3, b_2, b_1, d_2, d_1, but omits the list (5.4) of symmetrical models which are eliminated by `MaxRes`.

The next unsatisfiable core would have to be $\{\neg b_2 \vee \neg d_2, C_3, C_5, C_6\}$ to hit all rows in (5.7). Repeating the resolution process yields a problem ϕ^2 which is satisfiable except for the 2 *false* soft clauses. Therefore any model of the satisfiable problem ϕ^2 with $z = 0$ corresponds to a model of the original problem ϕ^0 with $z = 2$ = the iteration count. □

5.5.1 `MaxRes` algorithm – summary and analysis

To aid the analysis of `WPM1` we replaced the one-hot encoding $b_1, \ldots, b_n$ generated by `WPM1` with a new problem variable $j \in 1..n$. We make a similar argument about `MaxRes`.

We define ordered lists $B = [b_p, \ldots, b_1]$ and $D = [d_p, \ldots, d_1]$ of the objective variables and selector variables introduced by `MaxRes`, in the reverse order that they are resolved by *ApplyMaxRes*(). Then $d_i \leftrightarrow b_{i+1} \vee \ldots \vee b_p$ posted by *ApplyMaxRes*() can be stated in a more convenient left-to-right form as $D[i] = B[1] \vee \ldots \vee B[i-1]$.

Now consider a hypothetical new problem variable $j \in 1..p$ giving the index of the first nonzero objective term in B. Clearly $D[i] = \mathit{false}$ when $i \leq j$, since $B[1] = \ldots = B[j-1] = \mathit{false}$. And furthermore, clearly $D[i] = \mathit{true}$ when $i > j$, since $B[j] = \mathit{true}$.

Then D defines an order-encoding [107] of j, by $D[i] \equiv [j \leq i-1], i \in 1..p$. Now consider the soft compensation clauses $\neg b_i \vee \neg d_i$ posted by *ApplyMaxRes*(), equivalently $\neg B[i] \vee \neg D[i]$. We can rewrite these to the soft clauses $\neg B[i] \vee [j > i-1], i \in 1..p$.

In summary then, `MaxRes` replaces a singleton soft clause $\neg B[i]$ with a relaxed version as a soft clause $\neg B[i] \vee [j > i-1]$ where $j = \min\{k : B[k] = \mathit{true}\}$. This relaxes all

Algorithm 5.7 Post-core processing for `MaxRes` in our SAT-optimization framework

function *process_MaxRes*(w_{min}, $\{x_1, \ldots, x_n\}$)

1 let $a_1, \ldots, a_n$ = new literals
2 let $[j \leq 1], \ldots, [j \leq n-1]$ = an order-encoding of a new problem variable $j \in 1..n$
3 define convenience literals $[j \leq 0] = 0$ (*false*), $[j \leq n] = 1$ (*true*)
4 **return** $(\bigcup_{i=1}^{n} \{[j \leq i-1] \rightarrow [j \leq i], x_i \wedge [j \leq i-1] \rightarrow a_i, x_i \rightarrow [j \leq i]\},$ # C_{new}
5 $\{(w_{min}, a_i) : i \in 1..n\})$ # $objective_{new}$

singleton soft clauses $\neg B[1], \ldots, \neg B[j]$, although since soft clauses $\neg B[1], \ldots, \neg B[j-1]$ are not violated due to the definition of j, the net effect is to relax only $\neg B[j]$.

Therefore `MaxRes` is essentially the same as `WPM1` which also relaxes a single soft clause defined by the value of j. The difference is that `MaxRes` calculates j from the unsatisfiable core, rather than leaving it as a free variable to be assigned by the solver.

5.5.2 `MaxRes` algorithm – SAT-optimization version

Algorithm 5.7 shows the post-core processing for our `MaxRes` algorithm, as a backend subroutine *process_MaxRes*() which is identical to *process_WPM1*() except for lines 4–5, which encode the slightly different handling of the new problem variable j.

To emulate `MaxRes` we have to add the additional constraint $j = min\{i : x_i = true\}$. In the implementation we weaken '=' to '≤' in this additional constraint, since optimization ensures that $x_j = true$ as it already does for the `WPM1` version of the algorithm (recalling that at least one of $x_1, \ldots x_n$ must be *true*), giving the '≥' side of the constraint.

We encode the constraint $j \leq min\{i : x_i = true\}$ by clauses $x_i \rightarrow [j \leq i], i \in 1..n$. Since the `WPM1` version of the algorithm already had clauses $x_i \wedge [j > i] \rightarrow a_i$, equivalently $x_i \wedge \neg a_i \rightarrow [j \leq i]$, we just strengthened this latter by removing $\neg a_i$ from the LHS.

Example 5.6 Consider how Algorithm 5.1 with *algorithm* = `MaxRes` tackles the SAT-optimization problem (5.1). Just as in the `WPM1` example, the first SAT call is

$$C = \{C'_1, C'_2, C'_3, C'_4, C'_5, C'_6\}$$
$$A = \{v_1 \mapsto 1, v_2 \mapsto 1, v_3 \mapsto 1, v_4 \mapsto 1, v_5 \mapsto 1, v_6 \mapsto 1\}$$
$$SAT(D_{orig}, C', \{\neg v_1, \neg v_2, \neg v_3, \neg v_4, \neg v_5, \neg v_6\}) = unsatisfiable(\{\neg v_1, \neg v_2, \neg v_4\}),$$

with the resolution tree of Figure 5.1b. Hence we construct the next problem

$$
\begin{aligned}
C = \{&C'_1, C'_2, C'_3, C'_4, C'_5, C'_6,\\
&\cancel{\boldsymbol{v_1 \wedge [j \leq 0] \rightarrow a_1}}, \boldsymbol{v_1 \rightarrow [j \leq 1]},\\
&\boldsymbol{v_2 \wedge [j \leq 1] \rightarrow a_2}, \boldsymbol{v_2 \rightarrow [j \leq 2]},\\
&\boldsymbol{v_4 \wedge [j \leq 2] \rightarrow a_3}, \cancel{\boldsymbol{v_4 \rightarrow [j \leq 3]}}\}\\
A = \{&v_1 \mapsto \dot{+}\,\mathbf{0}, v_2 \mapsto \dot{+}\,\mathbf{0}, v_3 \mapsto 1, v_4 \mapsto \dot{+}\,\mathbf{0}, v_5 \mapsto 1, v_6 \mapsto 1,\\
&\boldsymbol{a_1 \mapsto 1}, \boldsymbol{a_2 \mapsto 1}, \boldsymbol{a_3 \mapsto 1}\}.
\end{aligned}
$$

where **boldface** and ~~strikeout~~ denote additions, removals or changes, and cancellation shows clauses which are always satisfied and hence not stored, analogously to the optimization performed by *ApplyMaxRes*() lines 4–8. Indeed there is no constraint which ever asserts a_1 in the above problem, but we show a_1 to make the construction explicit.

This has the same list (5.5) of non-dominated models as the problem generated by the first iteration of Algorithm 5.1 with *algorithm* = `WPM1`, omitting the list (5.6) of symmetrical models which are eliminated by `MaxRes`. However, similarly to *algorithm* = `WPM1`, but unlike the published `MaxRes` algorithm which calculates j exactly, there are some dominated models where j is smaller than necessary and does not encode any violation, e.g.

x_1	x_2	x_3	$\boldsymbol{j}$	~~v_1~~ $\boldsymbol{a_1}$	~~v_2~~ $\boldsymbol{a_2}$	v_3	~~v_4~~ $\boldsymbol{a_3}$	v_5	v_6	z
0	0	0	**1**	0	0	0	1	1	1	3.

The next unsatisfiable core would have to be $\{\neg a_2, \neg v_3, \neg v_5, \neg v_6\}$ to hit all rows of (5.5). Repeating the relaxation process gives $z_{lb} = 2$ and an optimal model. □

We claim our version of the `MaxRes` algorithm using the backend function *process_MaxRes*() given in this section, emulates the published `MaxRes` algorithm [79].

Theorem 5.2 *Our Algorithm 5.1 with algorithm =* `MaxRes` *on a SAT-optimization problem P, emulates the published Algorithms 5.4–5.6 on a MaxSAT problem Q, where P and Q are related by the SAT-optimization transformation given in Definition 5.1.*

Proof (sketch) Construct Algorithm 5.1′ from Algorithm 5.1 by posting $j = min\{i : x_i = true\}$ instead of just the '$\leq$' side. Show that Algorithms 5.1′ and 5.1 are equivalent.

Consider the problems P_k, Q_k produced by Algorithms 5.1′ and 5.4–5.6 after k iterations. Given that the theorem holds for P_{k-1}, Q_{k-1}, show that starting with P_{k-1}, Q_{k-1}

(i) Algorithm 5.5 locally applies the SAT-optimization transformation to the soft clauses in the unsatisfiable core, whereupon these clauses become hard clauses and thus equivalent in P_k and Q_k regardless of the SAT-optimization transformation;

(ii) both algorithms thus obtain an unsatisfiable core of literals $B[1..n]$ which cannot all be zero simultaneously, called $x_1, \ldots, x_n$ and $b_n, b_{n-1}, \ldots, b_1$ respectively;

Algorithm 5.8 The published OLL algorithm

Input: A formula φ

```
1  (φW, φS, φSC) ← (φ, Soft(φW), ∅)          # SIC: should be Soft(φ) not Soft(φW)
2  map ← ∅                                    # map(lit) = (sumOtps, bound)
3  while true do
4      (st, φC, A) ← SATSolver(φW)
5      if st = true then return Σ_{(c,1)∈φS} (1 − A(c))
6      else
7          (L, φW) ← RelaxAndHarden(φW, φC ∩ φS)
8          for each (¬s, 1) ∈ φC ∩ φSC do
9              φW ← φW \ {(¬s, 1)}
10             φSC ← φSC \ {(¬s, 1)}
11             L ← L ∪ {s}
12             (sumOtps, b) ← map(¬s)
13             if b + 1 < |sumOtps| then
14                 φW ← φW ∪ {(¬sumOtps[b + 1], 1)}
15                 φSC ← φSC ∪ {(¬sumOtps[b + 1], 1)}
16                 map(¬sumOtps[b + 1]) ← (sumOtps, b + 1)
17         (sumOtps_New, sumCls_New) ← createSum(L)
18         φW ← φW ∪ {(c, ⊤) : c ∈ sumCls_New} ∪ {(¬sumOtps_New[1], 1)}
19         φSC ← φSC ∪ {(¬sumOtps_New[1], 1)}
20         map(¬sumOtps_New[1]) ← (sumOtps_New, 1)
```

(iii) both algorithms calculate literals $D[1..n]$ giving the first nonzero element of $B[1..n]$, called $[j \leq 0], \ldots, j \leq n-1]$ and $\mathit{false}, d_{n-1} \ldots, d_1$ respectively;

(iv) Algorithm 5.1′ adds the SAT-optimization transform of, and Algorithm 5.6 adds directly, a set of soft clauses $\neg B[i] \vee \neg D[i], i \in 1..n$, in the form $x_i \wedge [j \leq i-1] \rightarrow a_i$ (where a_i is a new objective literal) and $\neg b_{n+1-i} \vee \neg d_{n+1-i}$ respectively;

to create P_k, Q_k which must be equivalent up to the SAT-optimization transformation. □

5.6 OLL unsatisfiable-core algorithm

Algorithm 5.8 shows the published OLL algorithm [77]. Given a set of soft or hard clauses, line 4 solves the corresponding SAT problem treating soft clauses as hard, obtaining an unsatisfiable core φ_C if this fails. Line 7 locally applies the SAT-optimization transformation of Definition 5.1 to just the soft clauses in the unsatisfiable core, obtaining a revised clause-set and a set of objective variables L which we denote $x_1, \ldots, x_n$. Initially $\varphi_{SC} = \emptyset$, so lines 8–16 (for scheduled addition of soft clauses) do not execute the first time.

Line 17 creates what is effectively a new integer variable $j \in 1..n$ and obtains an ordered list of literals $\mathit{sumOtps}_{New} = [[j > 1], \ldots, [j > n-1]]$ and a CNF encoding of a cardinality constraint $\mathit{sumCls}_{New} = \mathit{CNF}(j \geq x_1 + \ldots + x_n)$. It then adds a new soft clause $[j \leq 1]$, but this soft clause is placed in a special set φ_{SC}. As soft clauses in φ_{SC} become violated,

detected by the test at line 8 for these soft clauses appearing in an unsatisfiable core, lines 9–16 lazily add further soft clauses $[j \leq 2], [j \leq 3], \ldots, [j \leq n-1]$, to φ_{SC}.

Example 5.7 Consider how OLL tackles the MaxSAT problem (2.1), consisting of clauses

$$C_1 \equiv \neg x_1, \qquad C_2 \equiv \neg x_2, \qquad C_3 \equiv \neg x_3,$$
$$C_4 \equiv x_1 \vee x_2, \qquad C_5 \equiv x_1 \vee x_3, \qquad C_6 \equiv x_2 \vee x_3,$$

each with weight 1. The first SAT call is

$$\varphi_W = \{(C_1, 1), (C_2, 1), (C_3, 1), (C_4, 1), (C_5, 1), (C_6, 1)\}$$
$$SATSolver(\varphi_W) = (\mathit{false}, \{C_1, C_2, C_4\}, \mathit{undefined}),$$

with the resolution tree of Figure 5.1a. Thus the unsatisfiable core is $\varphi_C = \{C_1, C_2, C_4\}$. After locally applying the SAT-optimization transform, and then checking φ_{SC} which does nothing in this case, the problem to be relaxed at line 17 consists of

$$\varphi_W = \{(C_1 \vee \boldsymbol{b_1}, \boldsymbol{+\infty}), (C_2 \vee \boldsymbol{b_2}, \boldsymbol{+\infty}), (C_3, 1), (C_4 \vee \boldsymbol{b_3}, \boldsymbol{+\infty}), (C_5, 1), (C_6, 1)\}$$
$$L = \{b_1, b_2, b_3\},$$

where **boldface** denotes additions and ~~strikeout~~ denotes removals or changes relative to the first SAT call, and L gives an objective $b_1 + b_2 + b_3$ to minimize. Defining a new problem variable $j \in 1..3$ which is at least this objective, then encoding $j - 1$ into literals as $[j > 1] + [j > 2]$ and then into soft clauses $[j \leq 1], [j \leq 2]$ gives

$$\varphi_W = \{(C_1 \vee b_1, \infty), (C_2 \vee b_2, \infty), (C_3, 1), (C_4 \vee b_3, \infty), (C_5, 1), (C_6, 1),$$
$$\boldsymbol{([j \leq 1], 1), ([j \leq 2], 1)}\} \cup \boldsymbol{HARD(CNF(j \geq b_1 + b_2 + b_3))}.$$

The revised problem φ_W has non-dominated models

x_1	x_2	x_3	$\boldsymbol{b_1}$	$\boldsymbol{b_2}$	$\boldsymbol{b_3}$	$\boldsymbol{j}$	~~C_1~~	~~C_2~~	C_3	~~C_4~~	C_5	C_6	$\boldsymbol{[j \leq 1]}$	$\boldsymbol{[j \leq 2]}$	z
0	0	0	**0**	**0**	**1**	**1**	~~0~~	~~0~~	0	~~1~~	1	1	**0**	**0**	~~3~~ **2**
1	0	0	**1**	**0**	**0**	**1**	~~1~~	~~0~~	0	~~0~~	0	1	**0**	**0**	~~2~~ **1**
0	1	0	**0**	**1**	**0**	**1**	~~0~~	~~1~~	0	~~0~~	1	0	**0**	**0**	~~2~~ **1**
1	1	0	**1**	**1**	**0**	**2**	~~1~~	~~1~~	0	~~0~~	0	0	**1**	**0**	~~2~~ **1**
0	0	1	**0**	**0**	**1**	**1**	~~0~~	~~0~~	1	~~1~~	0	0	**0**	**0**	~~2~~ **1**
1	0	1	**1**	**0**	**0**	**1**	~~1~~	~~0~~	1	~~0~~	0	0	**0**	**0**	~~2~~ **1**
0	1	1	**0**	**1**	**0**	**1**	~~0~~	~~1~~	1	~~0~~	0	0	**0**	**0**	~~2~~ **1**
1	1	1	**1**	**1**	**0**	**2**	~~1~~	~~1~~	1	~~0~~	0	0	**1**	**0**	~~3~~ **2**.

(5.8)

Algorithm 5.9 Post-core processing for OLL in our SAT-optimization framework

function *process_OLL*(w_{min}, $\{x_1, \ldots, x_n\}$)
1 let $[j \leq 1], \ldots, [j \leq n-1]$ = new literals
2 **return** $(\bigcup_{i=1}^{n-1} CNF([j \leq i] \rightarrow \sum_{k=1}^{n} x_k \leq i), \{(w_{min}, [j > i]) : i \in 1..n-1\})$

There are also some dominated models where j is larger than $b_1 + b_2 + b_3$, e.g.

x_1	x_2	x_3	$\boldsymbol{b_1}$	$\boldsymbol{b_2}$	$\boldsymbol{b_3}$	$\boldsymbol{j}$	~~C_7~~	~~C_2~~	C_3	~~C_4~~	C_5	C_6	$\boldsymbol{[j \leq 1]}$	$\boldsymbol{[j \leq 2]}$	z
0	0	0	**0**	**0**	**1**	**2**	0	0	0	1	1	1	**1**	**0**	3.

Note that lines 18–20 would have placed $[j \leq 1]$ in φ_W, and scheduled $[j \leq 2]$ via φ_{SC} and *map* to add after $[j \leq 1]$ was proven to be violated, but we show both for clarity.

The next unsatisfiable core would have to be $\{C_3, C_5, C_6, [j \leq 1]\}$ to hit all rows in (5.8). Repeating the relaxation process gives a satisfiable problem. That is, the problem after 2 iterations has a model with $z = 0$, giving a model of the original problem with $z = 2$. □

5.6.1 OLL algorithm – summary and analysis

The OLL algorithm locally applies the SAT-optimization transformation to the unsatisfiable core to obtain a local objective $j = x_1 + \ldots + x_n$ to minimize, subtracts one from j to relax the problem, encodes this $j - 1$ into literals as $[j > 1] + \ldots + [j > n-1]$, and then re-encodes this into MaxSAT as singleton soft clauses, scheduling them for later addition.

5.6.2 OLL algorithm – SAT-optimization version

Algorithm 5.9 shows the post-core processing for our OLL algorithm.

Given an unsatisfiable core $\{\neg x_1, \ldots, \neg x_n\}$, which means that $x_1, \ldots, x_n$ cannot all be zero simultaneously, the backend subroutine receives the set $\{x_1, \ldots, x_n\}$, and a weight w_{min} such that any nonzero $x_i, i \in 1..n$ incurs a cost of w_{min} in the objective.

Note that $x_1, \ldots, x_n$ have been removed from the assumptions map A prior to calling the backend subroutine, or at least had their weight reduced by w_{min}. That is, the problem objective has been reduced by $w_{min}x_1 + \ldots + w_{min}x_n$. The backend subroutine has to add this back into the objective, but less w_{min} (recalling that $w_{min}x_1 + \ldots + w_{min}x_n \geq w_{min}$, otherwise $x_1, \ldots, x_n$ would not have been an unsatisfiable core), to relax the problem.

To partially reinstate the assumptions, line 1 defines a new problem variable $j \in 1..n$, via its encoding into literals $[j \leq 1], \ldots, [j \leq n-1]$. Then line 2 posts what is effectively a new constraint $j = x_1 + \ldots + x_n$ and a new objective term $w_{min}(j-1)$.

We post only the '$\geq$' side of the new constraint, since optimization will minimize j. We encode the resulting new constraint $j \geq x_1 + \ldots + x_n$ by a series of implications $[j \leq i] \rightarrow x_1 + \ldots + x_n \leq i, i \in 1..n-1$, via Abío et al.'s CNF encoding [1]. We encode the new objective term into Pseudo-Boolean by $w_{min}[j > 1] + \ldots + w_{min}[j > n-1]$.

Example 5.8 Consider how Algorithm 5.1 with *algorithm* = `OLL` tackles the SAT-optimization problem (5.1). Just as in the `WPM1`/`MaxRes` examples, the first SAT call is

$$C = \{C'_1, C'_2, C'_3, C'_4, C'_5, C'_6\}$$
$$A = \{v_1 \mapsto 1, v_2 \mapsto 1, v_3 \mapsto 1, v_4 \mapsto 1, v_5 \mapsto 1, v_6 \mapsto 1\}$$
$$SAT(D_{orig}, C', \{\neg v_1, \neg v_2, \neg v_3, \neg v_4, \neg v_5, \neg v_6\}) = \mathit{unsatisfiable}(\{\neg v_1, \neg v_2, \neg v_4\}),$$

with the resolution tree of Figure 5.1b. Hence we construct the next problem

$$\begin{aligned} C = \{&C'_1, C'_2, C'_3, C'_4, C'_5, C'_6\} \\ &\cup \mathit{CNF}([j \leq 1] \rightarrow v_1 + v_2 + v_4 \leq 1) \\ &\cup \mathit{CNF}([j \leq 2] \rightarrow v_1 + v_2 + v_4 \leq 2) \\ A = \{&v_1 \mapsto \cancel{1}\,\mathbf{0}, v_2 \mapsto \cancel{1}\,\mathbf{0}, v_3 \mapsto 1, v_4 \mapsto \cancel{1}\,\mathbf{0}, v_5 \mapsto 1, v_6 \mapsto 1, \\ &[j > 1] \mapsto \mathbf{1}, [j > 2] \mapsto \mathbf{1}\}. \end{aligned}$$

where **boldface** and ~~strikeout~~ denote additions or changes. This has non-dominated models

x_1	x_2	x_3	j	$\cancel{v_1}$	$\cancel{v_2}$	v_3	$\cancel{v_4}$	v_5	v_6	$[j > 1]$	$[j > 2]$	z
0	0	0	**1**	~~0~~	~~0~~	0	~~1~~	1	1	**0**	**0**	~~3~~ **2**
1	0	0	**1**	~~1~~	~~0~~	0	~~0~~	0	1	**0**	**0**	~~2~~ **1**
0	1	0	**1**	~~0~~	~~1~~	0	~~0~~	1	0	**0**	**0**	~~2~~ **1**
1	1	0	**2**	~~1~~	~~1~~	0	~~0~~	0	0	**1**	**0**	~~2~~ **1**
0	0	1	**1**	~~0~~	~~0~~	1	~~1~~	0	0	**0**	**0**	~~2~~ **1**
1	0	1	**1**	~~1~~	~~0~~	1	~~0~~	0	0	**0**	**0**	~~2~~ **1**
0	1	1	**1**	~~0~~	~~1~~	1	~~0~~	0	0	**0**	**0**	~~2~~ **1**
1	1	1	**2**	~~1~~	~~1~~	1	~~0~~	0	0	**1**	**0**	~~3~~ **2**.

(5.9)

There are also some dominated models where j is larger than $v_1 + v_2 + v_4$, e.g.

x_1	x_2	x_3	j	$\cancel{v_1}$	$\cancel{v_2}$	v_3	$\cancel{v_4}$	v_5	v_6	$[j > 1]$	$[j > 2]$	z
0	0	0	**2**	~~0~~	~~0~~	0	~~1~~	1	1	**1**	**0**	3.

Clearly the non-dominated models (5.9) of the SAT-optimization `OLL` algorithm after one iteration, are the same as the non-dominated models (5.8) of the published `OLL` algorithm, up to the different objective encoding (linear terms vs. soft clauses). The added b_1, b_2, b_3 in (5.8) result from locally applying the SAT-optimization transformation to the unsatisfiable core, equivalent to v_1, v_2, v_4 in (5.9) from globally applying the transformation.

The next unsatisfiable core would have to be $\{\neg v_3, \neg v_5, \neg v_6, [j \leq 1]\}$ to hit all rows of (5.9). Repeating the relaxation process gives $z_{lb} = 2$ and an optimal model. □

We claim our version of the OLL algorithm using the backend function *process_OLL*() given in this section, emulates the published OLL algorithm [77].

Theorem 5.3 *Our Algorithm 5.1 with algorithm = OLL on a SAT-optimization problem P, emulates the published Algorithm 5.8 on a MaxSAT problem Q, where P and Q are related by the SAT-optimization transformation given in Definition 5.1.*

Proof (sketch) Construct Algorithm 5.8′ from Algorithm 5.8 to add objective terms immediately instead of scheduling them for lazy addition, e.g. by moving lines 8–16 to appear after line 19 and changing $\varphi_C \cap \varphi_{SC}$ to just φ_{SC} so that all terms are added instead of only those in the unsatisfiable core. Show that Algorithms 5.8′ and 5.8 are equivalent.

Consider the problems P_k, Q_k produced by Algorithms 5.1 and 5.8′ after k iterations. Given that the theorem holds for P_{k-1}, Q_{k-1}, show that starting with P_{k-1}, Q_{k-1}

(i) Algorithm 5.8′ locally applies the SAT-optimization transformation to the soft clauses in the unsatisfiable core, whereupon these clauses become hard clauses and thus equivalent in P_k and Q_k regardless of the SAT-optimization transformation;
(ii) both algorithms thus obtain an unsatisfiable core of literals $x_1, \ldots, x_n$ which cannot all be zero simultaneously, referred to as the set L in Algorithm 5.8′;
(iii) both algorithms create a new variable $j \in 1..n$ defined by $j \geq x_1 + \ldots + x_n$ and decomposed to $[j > 0] + \ldots + [j > n-1]$, held in $sumOtps_{New}$ in Algorithm 5.8′;
(iv) Algorithm 5.1 adds new objective terms $[j > 1], \ldots, [j > n-1]$, whereas Algorithm 5.8′ adds new singleton soft clauses $[j \leq 1], \ldots, [j \leq n-1]$;

to create P_k, Q_k which must be equivalent up to the SAT-optimization transformation, except that applying the SAT-optimization transformation to the soft clauses $[j \leq i], i \in 1..n-1$ from step iv, will create a violation variable $[j > i]'$ and hard clause $[j > i] \rightarrow [j > i]'$, and put $[j > i]'$ in the objective, this has to be simplified out to show the equivalence. □

5.7 MSU3 unsatisfiable-core algorithm

Algorithm 5.10 shows the published MSU3 algorithm [71]. Given a set of soft clauses φ_W initialized at line 1, the algorithm executes 3 separate loops to completion in turn,

(i) at lines 2–8, solving φ_W treating soft clauses as hard, and deleting any clause which appears in an unsatisfiable core (collecting these in UC), until φ_W is satisfiable;
(ii) at lines 9–17, locally applying the SAT-optimization transformation of Definition 5.1 to the clauses in UC to define a new set of hard clauses and an objective, and adding these hard clauses back into φ_W in an identifiable manner;
(iii) at lines 18–31, interleaving the bodies of loops i and ii and updating the objective in each iteration, until φ_W is satisfiable under the assumption that the objective equals λ, where λ is the iteration count and also an objective lower bound (like i in Algorithm 5.4, since this version of MSU3 considers MaxSAT, not weighted MaxSAT).

Algorithm 5.10 The published MSU3 algorithm

```
1  φ_W ← φ                                          # Working formula, initially set to φ
2  UC ← ∅
3  while true do                                    # Phase 1: Identify disjoint cores
4      (st, φ_C) ← SAT(φ_W)                         # φ_C is an unsat core if φ_W is unsat
5      if st = UNSAT then
6          φ_W ← φ_W \ φ_C
7          UC ← UC ∪ {φ_C}
8      else break                                   # Move to 2nd loop
9  BV ← ∅
10 for each φ_C ∈ UC do                             # Add blocking variables
11     for each ω ∈ φ_C do
12         b is a new blocking variable
13         BV ← BV ∪ {b}
14         φ_W ← φ_W ∪ {ω ∪ {b}}
15 λ = |UC|                                         # Lower bound on true blocking variables
16 φ_B ← CNF(∑_{b∈BV} b = λ)
17 φ_W ← φ_W ∪ φ_B                                  # Current cardinality constraint
18 while true do                                    # Phase 2: Increment lower bound λ
19     (st, φ_C) ← SAT(φ_W)                         # φ_C is an unsat core if φ_W is unsat
20     if st = UNSAT then
21         λ ← λ + 1
22         for each ω ∈ φ_C do
23             if ω has no blocking variable then
24                 b is new blocking variable
25                 ω_B ← ω ∪ {b}                    # ω_B is tagged non-auxiliary
26                 φ_W ← φ_W \ {ω} ∪ {ω_B}
27                 BV ← BV ∪ {b}
28         φ_W ← φ_W \ φ_B
29         φ_B ← CNF(∑_{b∈BV} b = λ)                # New cardinality constraint
30         φ_W ← φ_W ∪ φ_B                          # Clauses in φ_B are tagged auxiliary
31     else return |φ| − λ                          # Solution to MaxSat problem
```

Example 5.9 Consider how MSU3 tackles the MaxSAT problem (2.1), consisting of clauses

$$C_1 \equiv \neg x_1, \qquad C_2 \equiv \neg x_2, \qquad C_3 \equiv \neg x_3,$$
$$C_4 \equiv x_1 \vee x_2, \qquad C_5 \equiv x_1 \vee x_3, \qquad C_6 \equiv x_2 \vee x_3,$$

each with weight 1. Loop i makes the first SAT call at line 4,

$$\varphi_W = \{C_1, C_2, C_3, C_4, C_5, C_6\} \qquad SAT(\varphi_W) = (\mathit{false}, \{C_1, C_2, C_4\}),$$

with the resolution tree of Figure 5.1a. Thus the unsatisfiable core is $\varphi_C = \{C_1, C_2, C_4\}$. After moving these clauses from φ_W to UC, loop i makes the second SAT call at line 4,

$$\varphi_W = \{C_3, C_5, C_6\} \qquad SAT(\varphi_W) = (\mathit{true}, \mathit{undefined}),$$

finding a model $[x_1, x_2, x_3] = [1, 1, 0]$. This model has $z = 2$, which is arbitrary and may be larger than the number of unsatisfiable cores found so far. Then loop ii at lines 9–14 locally applies the SAT-optimization transformation to the clauses in $UC = \{C_1, C_2, C_4\}$ giving a revised problem at line 15 prior to setting up for the start of loop iii,

$$\varphi_W = \{C_1 \vee \boldsymbol{b_1}, C_2 \vee \boldsymbol{b_2}, C_3, C_4 \vee \boldsymbol{b_3}, C_5, C_6\} \qquad BV = \{b_1, b_2, b_3\},$$

where **boldface** denotes additions relative to the first SAT call, and BV gives an objective $z = b_1 + b_2 + b_3$ to minimize. Assuming soft clauses C_3, C_5, C_6 hold and using this partially constructed objective z gives the following models for the problem at line 15,

$$\begin{array}{cccccccccccc} x_1 & x_2 & x_3 & \boldsymbol{b_1} & \boldsymbol{b_2} & \boldsymbol{b_3} & \cancel{C_1} & \cancel{C_2} & C_3 & \cancel{C_4} & C_5 & C_6 \quad z \\ 1 & 1 & 0 & \mathbf{1} & \mathbf{1} & \mathbf{0} & \cancel{1} & \cancel{1} & 0 & \cancel{0} & 0 & 0 \quad 2. \end{array} \tag{5.10}$$

Lines 15–17 add an objective constraint $\varphi_B = CNF(b_1 + b_2 + b_3 \leq 1)$ based on BV and $\lambda = 1$. Then loop iii executes starting with the third SAT call at line 19,

$$\varphi_W = \{C_1 \vee b_1, C_2 \vee b_2, C_3, C_4 \vee b_3, C_5, C_6\} \cup \boldsymbol{\varphi_B}$$
$$SAT(\varphi_W) = (\mathit{false}, \{C_3, C_5, C_6, \text{some subset of the hard clauses}\}),$$

Here C_3, C_5, C_6 cannot hold simultaneously because the unique model in the list (5.10) was cut off by φ_B. Thus at least some of the hard clauses in φ_B would have to be in the unsatisfiable core as well, but we assume the test at line 23 is supposed to ignore these.

Lines 22–27 transform C_3, C_5, C_6 adding further blocking variables b_4, b_5, b_6 and updating the objective to $z = b_1 + \ldots + b_6$, completing the SAT-optimization transformation. Lines 28–30 then remove φ_B from φ_W and updates it to $\varphi_B = CNF(b_1 + \ldots + b_6 \leq 2)$ based on $\lambda = 2$. This relaxation allows the fourth SAT call to find any model with $z = 2$ from the list (5.2) of models for the SAT-optimization version (5.1) of the MaxSAT problem (2.1) (ignoring the renaming of $b_1, \ldots, b_6$ to some permutation of $v_1, \ldots, v_6$). □

5.7.1 `MSU3` algorithm – summary and analysis

Loops i–ii are what we call the *calculate bounds* phase. As the authors observe [71], loop iii is capable of solving the problem directly, whereas loops i–ii are claimed to be an optimization to quickly derive a set of unsatisfiable cores. This serves two purposes, it increases the objective lower bound λ to some reasonable estimate before adding the clauses φ_B which implement the objective (thus saving some overhead), and it proves the problem satisfiable giving some

reasonable estimate of the objective upper bound (z_{ub} in Algorithm 5.1), although MSU3 as given here does not calculate or use the upper bound information.

We conceptualize MSU3 as an enhanced version of plain LSUS (simple increasing-objective optimization) in which a special assumption constrains the objective to a value that increases by 1 on each unsatisfiable attempt. The enhancements over plain LSUS are (i) the calculate bounds phase, although its usefulness isn't established, (ii) building the objective lazily, (iii) aggressively assuming soft clauses hold, and (iv) just as aggressively backing off these assumptions, by removing/relaxing *all* clauses of each unsatisfiable core, leaving any optimization amongst these to the existing LSUS framework.

Example 5.10 To support this conceptual view, consider the previous Example 5.9. The first iteration of loop i transforms the soft clauses C_1, C_2, C_4, whereupon they become hard and can no longer appear in an unsatisfiable core. The first iteration of loop iii transforms the remaining soft clauses C_3, C_5, C_6 similarly. This gives effectively the SAT-optimization version (5.1) of the original MaxSAT problem (2.1). At this point, conceptually we could discover a series of empty unsatisfiable cores each of which increases λ by 1 until the objective constraint φ_B slackens enough to make the problem satisfiable.

We now suspect it is impossible to construct a problem where empty unsatisfiable cores are discovered and the algorithm reverts to plain LSUS, but it is a useful thought experiment, which illustrates the approach we take in our version of MSU3 in the next section.

5.7.2 MSU3 algorithm – SAT-optimization version

Our combined Algorithm 5.1 implements MSU3. Given the algorithm already has the assumptions map A to support MSU1/WPM1, MaxRes, OLL, and also the special assumption a to support LSUS, it was easy to put together a version of MSU3. We also found that our version naturally supports weighted partial MaxSAT rather than just MaxSAT.

Recall the parts of Algorithm 5.1, (i) applying queued problem modifications, (ii) solving the SAT problem under assumptions, (iii) if satisfiable constraining the objective, or (iv) if unsatisfiable removing the conflicting assumptions from A, then partially reinstating those assumptions depending on the algorithm in use (via the modification queue).

For MSU3, part iv removes all conflicting assumptions as normal, but there is no need to partially reinstate them – any objective term that has ever appeared in an unsatisfiable core is removed permanently from the assumptions map A. Since in our version we do not construct the objective lazily, this objective term is already in the objective assumption a. So MSU3 does not use a backend, or more correctly its backend is a no-operation.

In order to emulate the published MSU3 we had to implement a calculate-bounds phase. This is why we separated parts iv and i of Algorithm 5.1 and made them communicate via the variable *queue*. If *calc_bounds* = *false* then part iv is immediately followed by part i and unsatisfiable cores are processed directly. If *calc_bounds* = *true* then part iv

executes multiple times until the problem becomes satisfiable, and then part i executes once for each part iv, and thereafter parts i and iv are interleaved. This emulates the published Algorithm 5.10 without the duplication of the bodies of their loops i–ii in their loop iii. Note that our calculate-bounds phase works for any chosen *algorithm*.

Example 5.11 Consider how Algorithm 5.1 with *algorithm* = `MSU3` and *calc_bounds* = *true* tackles the SAT-optimization problem (5.1). The first SAT call is

$$C = \{C'_1, C'_2, C'_3, C'_4, C'_5, C'_6\}$$
$$A = \{v_1 \mapsto 1, v_2 \mapsto 1, v_3 \mapsto 1, v_4 \mapsto 1, v_5 \mapsto 1, v_6 \mapsto 1\}$$
$$SAT(D_{orig}, C', \{\neg v_1, \neg v_2, \neg v_3, \neg v_4, \neg v_5, \neg v_6\}) = \mathit{unsatisfiable}(\{\neg v_1, \neg v_2, \neg v_4\}),$$

with the resolution tree of Figure 5.1b. This is an unsatisfiable core with $w_{min} = 1$, so part iv at lines 27–37 executes to set $z_{lb} = 1$ and remove v_1, v_2, v_4 from A giving the next problem

$$C = \{C'_1, C'_2, C'_3, C'_4, C'_5, C'_6\}$$
$$A = \{v_1 \mapsto +\mathbf{0}, v_2 \mapsto +\mathbf{0}, v_3 \mapsto 1, v_4 \mapsto +\mathbf{0}, v_5 \mapsto 1, v_6 \mapsto 1\}$$
$$SAT(D_{orig}, C', \{\neg v_3, \neg v_5, \neg v_6\}) = \mathit{satisfiable}($$
$$\{x_1 \mapsto 1, x_2 \mapsto 1, x_3 \mapsto 0, v_1 \mapsto 1, v_2 \mapsto 1, v_3 \mapsto 0, v_4 \mapsto 1, v_5 \mapsto 0, v_6 \mapsto 0\}).$$

This is a model with $z = 2$, so part iii at lines 18–26 executes to save the model in *res* as the best found so far, set $z_{ub} = 2$, constrain the future objective to $< z_{ub} = 2$, and set *calc_bounds* = *false*. Since *calc_bounds* = *false*, part i at lines 8–18 executes to add the special assumption a constraining the objective to $\leq z_{lb} = 1$, giving the next problem

$$C = \{C'_1, C'_2, C'_3, C'_4, C'_5, C'_6\}$$
$$\cup\, \mathbf{CNF(v_1 + \ldots + v_6 < 2)}$$
$$\cup\, \mathbf{CNF(\neg a \rightarrow v_1 + \ldots + v_6 \leq 1)}$$
$$A = \{v_1 \mapsto 0, v_2 \mapsto 0, v_3 \mapsto 1, v_4 \mapsto 0, v_5 \mapsto 1, v_6 \mapsto 1, \mathbf{a \mapsto 1}\}$$
$$SAT(D_{orig}, C', \{\neg v_3, \neg v_5, \neg v_6\}) = \mathit{unsatisfiable}(\emptyset),$$

where **boldface** denotes additions relative to the previous SAT call. The constraint $v_1 + \ldots + v_6 < 2$ added at the end of the calculate-bounds phase cuts off all solutions regardless of assumptions. So the test at line 29 returns the best model found so far, with $z = 2$. □

We claim our version of the `MSU3` algorithm given in this section, approximately emulates the published `MSU3` algorithm [71]. A formal proof would be difficult due to our somewhat different approach, and since our termination rules may reduce iteration count compared with the published `MSU3`. So we omit this and refer to our experiments showing that both versions are comparable in performance, despite implementation differences.

Algorithm 5.11 The published `MSU4` algorithm

```
1  # Clauses of CNF formula φ are the initial clauses
2  φ_W ← φ                                  # Working formula, initially set to φ
3  μ_BV ← |φ|                               # Min blocking variables w/ value 1
4  ν_U ← 0                                  # Iterations w/ unsat outcome
5  V_B ← ∅                                  # IDs of blocking variables
6  UB ← |φ| + 1                             # Upper bound estimate
7  LB ← 0                                   # Lower bound estimate
8  while true do
9      (st, φ_C) ← SAT(φ_W)
10     # φ_C is an unsat core if φ_W is unsat
11     if st = UNSAT
12     then
13         φ_I ← φ_C ∩ φ                    # Initial clauses in core
14         I ← {i : ω_i ∈ φ_I}
15         V_B ← V_B ∪ I
16         if |I| > 0 then
17             φ_N ← {ω_i ∪ {b_i} : ω_i ∈ φ_I}
18             φ_W ← (φ_W \ φ_I) ∪ φ_N
19             φ_T ← CNF(∑_{i∈I} b_i ≥ 1)
20             φ_W ← φ_W ∪ φ_T
21         else # Solution to MaxSAT problem
22             return UB
23         ν_U ← ν_U + 1
24         UB ← |φ| − ν_U                   # Refine UB
25     else
26         ν ← |blocking variables w/ value 1|
27         if μ_BV < ν then
28             μ_BV ← ν
29             LB ← |φ| − μ_BV              # Refine LB
30         φ_T ← CNF(∑_{i∈V_B} b_i ≤ μ_BV − 1)
31         φ_W ← φ_W ∪ φ_T
32     if LB = UB then # Solution to MaxSAT problem
33         return UB
```

5.8 `MSU4` unsatisfiable-core algorithm

Algorithm 5.11 shows the published `MSU4` algorithm [72]. Given a set of soft clauses φ_W initialized at line 2, the algorithm executes a loop consisting of 3 parts,

(i) at line 9, solving φ_W treating soft clauses as hard;

(ii) if unsatisfiable, at lines 13–16 and 21–22 housekeeping by generating a set φ_I of soft clauses in the unsatisfiable core, giving them identifying numbers in I and checking for special cases, at lines 17–20 locally applying the SAT-optimization transformation of Definition 5.1 to the clauses in φ_I to define a new set of hard clauses and an objective which is updated incrementally, adding these hard clauses back into φ_W in

an identifiable manner, and at lines 23–24 updating an objective lower bound ν_U (we ignore the corresponding *UB* since it refers to satisfaction not violations); or

(iii) if satisfiable, at lines 26–29 updating an objective upper bound μ_{BV} (we ignore the corresponding *LB* since it refers to satisfaction not violations), and at lines 30–31 constraining the current objective so that it must improve next iteration.

The algorithm terminates if either the hard clauses of the problem become infeasible, or the lower bound ν_U meets the upper bound μ_{BV} (equivalently *UB* meets *LB*).

Example 5.12 Consider how `MSU4` tackles the MaxSAT problem (2.1), consisting of clauses

$$C_1 \equiv \neg x_1, \qquad C_2 \equiv \neg x_2, \qquad C_3 \equiv \neg x_3,$$
$$C_4 \equiv x_1 \vee x_2, \qquad C_5 \equiv x_1 \vee x_3, \qquad C_6 \equiv x_2 \vee x_3,$$

each with weight 1. Part i makes the first SAT call at line 9,

$$\varphi_W = \{C_1, C_2, C_3, C_4, C_5, C_6\} \qquad SAT(\varphi_W) = (\mathit{false}, \{C_1, C_2, C_4\}),$$

with the resolution tree of Figure 5.1a. Then part ii at lines 17–20 locally applies the SAT-optimization transformation to the clauses in $\varphi_C = \{C_1, C_2, C_4\}$ giving the next problem

$$\varphi_W = \{C_1 \vee \boldsymbol{b_1}, C_2 \vee \boldsymbol{b_2}, C_3, C_4 \vee \boldsymbol{b_3}, C_5, C_6\} \qquad BV = \{b_1, b_2, b_3\},$$

where **boldface** denotes additions relative to the first SAT call, and *BV* gives an objective $z = b_1 + b_2 + b_3$ to minimize. Lines 23–24 then update ν_U to record that the objective $z \geq \nu_U = 1$. Assuming soft clauses C_3, C_5, C_6 hold and using this partially constructed objective z gives the following models for the problem after one iteration,

x_1	x_2	x_3	$\boldsymbol{b_1}$	$\boldsymbol{b_2}$	$\boldsymbol{b_3}$	~~C_1~~	~~C_2~~	C_3	~~C_4~~	C_5	C_6	z
1	1	0	**1**	**1**	**0**	+	+	0	0	0	0	2.

Part i at line 9 makes the second SAT call and finds the above, unique, model. Then part iii at lines 26–29 update μ_{BV} to $z = 2$, and lines 30–31 enter a new objective constraint $b_1 + b_2 + b_3 \leq 1$. Finally, part i at line 9 makes the third SAT call and finds this revised problem infeasible regardless of any assumptions, so part ii at lines 21–22 terminates solving. We think perhaps the value returned at line 22 should be calculated from μ_{BV} (objective of the best solution found so far) rather than ν_U as in the published algorithm. □

5.8.1 `MSU4` algorithm – summary and analysis

We conceptualize `MSU4` as an enhanced version of plain `LSSU` (simple branch-and-bound optimization) in which an objective constraint added at each iteration gives better and better solutions until this becomes impossible. The enhancements over plain `LSSU` are (i) building

the objective lazily, (ii) aggressively assuming soft clauses hold, and (iii) just as aggressively backing off these assumptions, by removing/relaxing *all* clauses of each unsatisfiable core, leaving any optimization amongst these to the existing LSSU framework.

5.8.2 MSU4 algorithm – SAT-optimization version

Our combined Algorithm 5.1 implements MSU4. Given the algorithm already has the assumptions map A to support MSU1/WPM1, MaxRes, OLL, and also the objective constraints to support LSSU, it was easy to put together a version of MSU4. We also found that our version naturally supports weighted partial MaxSAT rather than just MaxSAT.

Our MSU4 algorithm is essentially our MSU3 algorithm without the special assumption a which constrains the objective to $\leq z_{lb}$, thus any model with objective $< z_{ub}$ can be found (initially infinite). So the same comments apply, except to say that part iii (finding a model) decreases z_{ub}, part iv (proving unsatisfiability) increases z_{lb}, and these interleave arbitrarily, exactly as in Algorithm 5.11 where part iii increases μ_{BV} and part ii decreases ν_U.

Example 5.13 Consider how Algorithm 5.1 with *algorithm* = MSU4 tackles the SAT-optimization problem (5.1). Solving proceeds the same as Example 5.11 except that the last SAT call omits the redundant constraint $\neg a \rightarrow v_1 + \ldots + v_6 \leq 1$ and associated assumption a, which had no effect since z_{lb} had already met z_{ub} in this simplified example anyway. □

Note that our MSU4 is like our MSU3 but always treating *calc_bounds* as *true*. Equivalently MSU3 with calculate-bounds means running as MSU4 until the first solution is found.

We claim our version of the MSU4 algorithm given in this section, emulates the published MSU3 algorithm [71]. The proof is similar to that of Theorem 5.3 (OLL), showing that both algorithms carry out parallel steps although the SAT-optimization transformation is performed in advance in our algorithm vs. progressively in the published algorithm.

5.9 Generalizing SAT-optimization to LCG

We now consider a generalization of the clause-splitting technique discussed in Section 5.2.1, equivalently the assumption-splitting technique discussed in Section 5.2.4.

A simple generalization in an LCG solver is to decompose an objective term cx where $x \in 0..m$, to $\sum_{i=0}^{m-1} c[x > i]$, making use of the bounds literals generated by the LCG solver for each integer variable and treating a literal as an integer 0..1. Disregarding the CP constraints in the problem (which LCG will convert to SAT lazily), this yields a SAT-optimization problem of the type shown in Example 5.1. Indeed, in our experiments comparing CP with MaxSAT solvers, we use this exact decomposition as part of the conversion of CP problems into equivalent weighted MaxSAT problems for the MaxSAT solvers.

Decomposition of the objective into literals generates a large overhead, particularly when the domains of the objective variables are large. This occurs, for example, in the RCPSP/WET

problems in Section 5.11.1, where an objective variable takes a value 0..*horizon* where *horizon* is the number of time steps considered. Since an assumption $[x \leq i]$ dominates $[x \leq j]$ when $i < j$, it would be better if we didn't need to maintain a 'remaining weight' for both assumptions, and nor do we wish to send large collections of assumptions to the LCG solver or receive unsatisfiable cores containing redundant assumptions.

We next define a scheme which keeps the dominated assumptions in implicit form, and calculates the 'remaining weight' and 'minimum weight' in a more intuitive way.

5.9.1 Basic LCG-optimization algorithm

Given an objective $c_1x_1 + \ldots + c_nx_n$ we define a new objective $y_1 + \ldots + y_n$ where $y_i = c_ix_i, i \in 1..n$. We then redefine the assumptions map A, relative to Algorithm 5.1, to contain an upper bound per objective term (rather than a remaining weight), initially 0, or $c\,\mathrm{lb}(x)$ when x is a CP variable with an arbitrary lower bound, perhaps negative. Then the assumptions set to send to the LCG solver can be determined from A as $\{[y_i \leq A(y_i)] : i \in 1..n\}$.

The revised objective completely removes weights from consideration since $y_i, i \in 1..n$ now have a compatible scale, so unsatisfiable cores containing literals $[y_i > a_i]$ are already normalized, and the literals are of equivalent status when sent to a backend such as *process_WPM1*(). That is, the unweighted MaxSAT algorithms can be used on the revised problem, on the basis that relaxing an assumption in unweighted MaxSAT, is advancing the assumed upper bound of an objective term y_i by 1 in CP (or possibly a larger increment, as we'll explain when we generalize w_{min} to LCG in the next section).

5.9.2 Implementing LCG-optimization using views

In the generalization to CP, the new objective term y_i replaces the original term $c_ix_i, i \in 1..n$. One way to map y_i to c_ix_i with an explicit *linear* or *times* constraint, $y_i = c_i \times x_i$. A more efficient approach is, since LCG uses bounds literals $[x_i \leq v]$ where v is a constant in the range $\mathrm{lb}(x_i)..\,\mathrm{ub}(x_i)$, we can simply map the bounds literals $[y_i \leq v]$ to $[x_i \leq \lfloor v/c_i \rfloor]$, where v and v/c_i are constant. This defines y_i as a view of c_ix_i with no overhead.

In the returned unsatisfiable core we take $[x_i > a_i]$ to mean $[y_i \geq c_i(a_i + 1)]$. Then $w_i = c_i(a_i+1) - A(y_i)$ is the smallest increment to $A(y_i)$ that relaxes $[x_i \leq a_i]$. Any increment less than $w_{min} = \min_{i=1}^{n} w_i$ relaxes nothing and therefore yields the same unsatisfiable core again. So this w_{min} can play the same role as the w_{min} in Algorithm 5.1.

5.10 Common LCG-optimization framework

Algorithm 5.12 shows our generalization of the core-guided MaxSAT algorithms from MaxSAT to CP. It is essentially similar to Algorithm 5.1 except that C is considered to

Algorithm 5.12 Common framework for LCG optimization

inputs:

D_{orig} = initial finite integer domains of variables
C = set of constraints $\{C_1, \ldots, C_n\}$ where C_i = any constraint with an LCG propagator
$objective = \{(c_1, x_1), \ldots, (c_n, x_n)\}$ with c_i a coefficient, x_i an integer CP variable
$algorithm \in \{\texttt{LSSU}, \texttt{LSUS}, \texttt{WPM1}, \texttt{MaxRes}, \texttt{OLL}, \texttt{MSU3}, \texttt{MSU4}\}$
$calc\_bounds$ = whether to identify disjoint unsatisfiable cores first

outputs:

$optimum\_found(S, z)$ if solution S of weight z found, or *unsatisfiable*

function *LCG_optimization*($C, objective, algorithm, calc\_bounds$)

$queue \leftarrow []$
if $algorithm \in \{\texttt{LSSU}, \texttt{LSUS}\}$ **then** $A \leftarrow \emptyset$ **else** $A \leftarrow \{x \mapsto (c\,\mathrm{lb}(x), c) : (c, x) \in objective\}$
$a_{lb} \leftarrow none$
$z_{lb} \leftarrow 0$
$z_{ub} \leftarrow \infty$
$res \leftarrow unsatisfiable$
while *true* **do**
  if $\neg calc\_bounds$ **then**
    for $(w_{min}, U) \in queue$ **do**
      case *algorithm* **in**
      `WPM1`: $(C_{new}, objective_{new}) \leftarrow process\_WPM1(w_{min}, U)$
      `MaxRes`: $(C_{new}, objective_{new}) \leftarrow process\_MaxRes(w_{min}, U)$
      `OLL`: $(C_{new}, objective_{new}) \leftarrow process\_OLL(w_{min}, U)$
      $(C, A) \leftarrow (C \cup C_{new}, A \cup \{x \mapsto (c\,\mathrm{lb}(x), c) : (c, x) \in objective_{new}\})$
    $queue \leftarrow []$
    if $algorithm \in \{\texttt{LSUS}, \texttt{MSU3}\}$ **then**
      let $a_{lb} \leftarrow$ new literal
      $(C, A) \leftarrow (C \cup \{\neg a_{lb} \rightarrow \sum_{(c,x) \in objective} cx \leq z_{lb}\}, A \cup \{a_{lb} \mapsto (0, 1)\})$
  # defer to LCG oracle, passing hard constraints and assumption literals $[x_i \leq a_i]$
  case ***LCG***$(D_{orig}, C, \{[x \leq \lfloor a/c \rfloor] : x \mapsto (a, c) \in A\})$ **in**
  satisfiable(S): # S = model as map from integer variable to integer value
    $z_{ub} \leftarrow \sum_{(c,x) \in objective} cS(x)$
    $res \leftarrow optimum\_found(S, z_{ub})$
    if $z_{ub} \leq z_{lb}$ **then return** *res*
    $C \leftarrow C \cup \{\sum_{(c,x) \in objective} cx < z_{ub}\}$
    $calc\_bounds \leftarrow false$
  unsatisfiable(U): # U = set of literals $[x_i \leq a_i]$ that cannot hold simultaneously
    $U \leftarrow \{\neg x : x \in U\}$ # want set of literals $[x_i > a_i]$ where one must be nonzero
    if $U = \emptyset$ **then return** *res*
    $w_{min} \leftarrow \min_{[x>a] \in U}(\mathrm{snd}(A(x))(a + 1) - \mathrm{fst}(A(x)))$
    $z_{lb} \leftarrow z_{lb} + w_{min}$
    if $z_{lb} \geq z_{ub}$ **then return** *res*
    if $algorithm \in \{\texttt{LSUS}, \texttt{MSU3}\}$ **and** $a_{lb} \neq none$ **then** $U \leftarrow U \cup \{a_{lb}\}$
    for $x \in U$ **do** $A(x) \leftarrow (\mathrm{fst}(A(x)) + w_{min}, \mathrm{snd}(A(x)))$
    if $algorithm \in \{\texttt{WPM1}, \texttt{MaxRes}, \texttt{OLL}\}$ **and** $|U| > 1$ **then**
      $queue \leftarrow queue + [(w_{min}, U)]$

Algorithm 5.13 Post-core processing for OLL in our LCG-optimization framework

function *process_OLL*(w_{min}, $\{x_1, \ldots, x_n\}$)
  let $y \in 1..n$ be a new integer CP variable
  return $(\{y \geq \sum_{i=1}^{n} x_i\}, \{(w_{min}, y)\})$

contain LCG constraints rather than clauses, and problem variables are considered to be integer rather than 0..1. In particular, *objective* is an integer linear combination.

In this version of the algorithm, we generalize the assumptions map A to contain, for each integer variable x in the objective, a pair (a, c), where c is the original coefficient of x in the objective, and a is the current assumed upper bound of the conceptual integer variable $y = cx$. We define $\text{fst}(A(x)) = a, \text{snd}(A(x)) = c$ where $x \mapsto (a, c) \in A$.

Lines 2 and 14 are modified to initialize $A(x)$ to $(c\,\text{lb}(x), c)$ for each objective term cx. Conceptually this means the upper bound of y is $\text{lb}(y)$ where $y = cx$. Thus each time some x is added to the objective, initial solving attempts will set x to its minimum value, which minimizes the objective assuming $c > 0$ (otherwise x can be negated). Then x can only increase from its minimum value if $\text{fst}(A(x))$ increases by at least c, which can only occur after x has appeared in at least one, possibly several, unsatisfiable cores.

Line 20 is modified to calculate the assumption to send to the SAT solver for each variable in the objective. If y should have an upper bound of $\text{fst}(A(x))$, we calculate the corresponding upper bound on x by floored division by c, or indeed $\text{snd}(A(x))$ which contains c.

Line 30 is modified to calculate w_{min} as the minimum increase to the upper bounds of all of the y variables appearing in an unsatisfiable core, such that at least one of the underlying x variables will have its corresponding upper bound increase by one.

Line 34 is modified to increase the upper bounds of all y variables by w_{min} before calling any backend function. Note that in the original Algorithm 5.1, the weight $A(x)$ started at c and had to be reduced to 0 before x was freed to increase by one step, and no further decrease was possible since x could only have domain of 0..1. Whereas here, the upper bound $\text{fst}(A(x))$ starts at 0 and has to be increased to c before x is freed to increase by one step, but further increase is possible too. We used a countdown in Algorithm 5.1 because it was intuitive given the existing practice of splitting clauses for weighted MaxSAT.

5.10.1 OLL algorithm – LCG-optimization version

Whilst the original backend Algorithms 5.3, 5.7, and 5.9 work correctly in the CP framework, the OLL backend contains a SAT decomposition which is unnecessary for LCG.

Algorithm 5.13 shows a more efficient backend, to be called from Algorithm 5.12. It shares the same calling interface as previously, noting that we treat integer CP variables interchangeably with SAT literals, on the basis that the latter are interpreted as 0..1.

The revised backend defines a new objective variable y as the sum of the literals passed in, and then arranges for $w_{min}y$ to be added in to the objective, with an initial assumption $y \leq 1$, which can be relaxed up to $y \leq n$ (that is, no assumption) as needed.

Example 5.14 Consider the CP optimization problem

$$\text{find } x_1, x_2 \in 0..10 \text{ minimizing } 2x_1 + 3x_2 \text{ such that } x_1 x_2 \geq 2.$$

The first two unsatisfiable cores are singletons and simply advance the assumptions (initially 0) so that $z_{lb} = 5$. The first non-trivial problem yields a further unsatisfiable core,

$$C = \{x_1 x_2 \geq 2\},$$
$$A = \{x_1 \mapsto (0\,\mathbf{2}, 2), x_2 \mapsto (0\,\mathbf{3}, 3)\},$$
$$LCG(D_{orig}, C, \{[x_1 \leq 1], [x_2 \leq 1]\}) = \mathit{unsatisfiable}([x_1 \leq 1], [x_2 \leq 1]).$$

Setting $x_1 = 2$ would require the assumed upper bound of $2x_1$ to advance from 2 to 4 hence $w_1 = 2$. Setting $x_2 = 2$ would require the assumed upper bound of $3x_2$ to advance from 3 to 6 hence $w_2 = 3$. Then $w_{min} = 2$, so this latter upper bound really advances from 3 to 5, and rounding makes the assumption $[x_2 \leq 1]$ stays in place for the next LCG call.

Compensating for these advances, Algorithm 5.12 increases z_{lb} to 7 and Algorithm 5.13 implements a new variable $y \in 1..2$ which charges an extra cost of 2 if $[x_1 > 1]$ and $[x_2 > 1]$ simultaneously, a situation which is assumed not to occur for the time being.

The next LCG call yields an optimal model with objective $z_{lb} = 7$,

$$C = \{x_1 x_2 \geq 2, \boldsymbol{y \geq [x_1 > 1] + [x_2 > 1]}\},$$
$$A = \{x_1 \mapsto (\cancel{2}\,\mathbf{4}, 2), x_2 \mapsto (\cancel{3}\,\mathbf{5}, 3), \boldsymbol{y \mapsto (2, 2)}\},$$
$$LCG(D_{orig}, C, \{[x_1 \leq 2], [x_2 \leq 1], [y \leq 1]\}) = \mathit{satisfiable}(\{x_1 \mapsto 2, x_2 \mapsto 1, y \mapsto 1\}). \quad \square$$

5.11 Experiments

In these experiments we evaluate our algorithms on a variety of well-known problems from the literature. We choose problems that benefit from unsatisfiable cores and also Constraint Programming (CP) and/or Lazy Clause Generation (LCG), yet which have a reasonable MaxSAT decomposition for comparison with the MaxSAT solvers.

We compare our unsatisfiable-core enabled LCG solver *LCG-glucose* implementing Algorithm 5.12, with *algorithm* = `WPM1`, `MaxRes`, `OLL`, `MSU3`, `LSSU`, `MSU4` and `LSUS`, against the best known publicly available MaxSAT implementation of each algorithm. These are (i) `MSU1/3/4` implemented in *MSUnCore* [70]; (ii) `MaxRes` implemented in *eva500a* [79]; (iii) `OLL` implemented in *MSCG* [77]; and (iv) `LSSU` implemented in *clasp* [51].

We insert a control comparison with our own MaxSAT solver *WCNF-glucose* implementing Algorithm 5.1, to isolate the effect of moving from LCG to MaxSAT (*LCG-glucose* vs. *WCNF-glucose*) as compared with the effect of algorithmic or implementation differences (*WCNF-glucose* vs. *WCNF-original* meaning *clasp*, *MSUnCore*, etc).

For the MaxSAT solvers, global constraints are decomposed using the MiniZinc standard library, and then the resulting integer model is encoded into SAT using an order-encoding [107] for *linear* constraints or other bounds-consistent constraints, a one-hot encoding [17] for *element* or other domain-consistent constraints, and in some cases both encodings linked as per Ohrimenko et al. [83]. *linear* constraints use Abío et al.'s [1] encoding, and other primitive constraints use our own best known encodings.

In addition to the MaxSAT solvers (our own and the original published versions), we compare with (i) *toulbar2* [96], an extensional solver using `LSSU` with cost shifting, expected to be comparable with `MSU4`, and (ii) *Z3-MaxSMT* [18], an SMT solver using `MaxRes`, expected to perform better than the equivalent MaxSAT solver *eva500a*.

The additional solvers run the same model as the MaxSAT solvers, except that they understand integer variables, so they use integer variables where possible. *linear* uses an a decomposition to binary operations (*toulbar2*) or the solver's native implementation (*Z3*). *element* uses extensional tables (*toulbar2*) or our usual SAT decomposition (*Z3*). Note that *Z3* uses a MaxSMT objective (identical to the MaxSAT solvers), which is a requirement for its `MaxRes` backend. Although *Z3*, like *LCG-glucose*, can understand an integer linear objective, it can only solve such problems using `LSSU`, not `MaxRes`.

We try several different decomposition approaches to the `OLL` algorithm. We try (i) Algorithm 5.9, denoted 'eager', (ii) a lazy variant of Algorithm 5.9, denoted 'lazy', which is equivalent except that dominated assumptions and associated constraints are not added until the dominating assumption gets relaxed, and (iii) Algorithm 5.13, denoted 'LCG'. Option ii emulates *MSCG* except that *MSCG* unlike *WCNF-glucose* reuses decomposition literals across invocations of *CNF*(). Note that *CNF*() is identity in *LCG-glucose*.

All solvers use their default search and restart strategies. In particular, *LCG-glucose* and *WCNF-glucose*, being modified versions of *glucose* 3.0 [10, 40], use an identical strategy which is *glucose*'s default. We simplify the generated *WCNF* (MaxSAT) models a priori using *glucose*'s simplifier, as some solvers use simplification and some do not.

We run on Intel dual processor × quad core Xeon E5405 nodes at 2.0 GHz with 16 Gb RAM per node. All solver executables are single-threaded and each core runs independently as a virtual single-threaded CPU, but we never schedule two different solvers simultaneously on the same physical node. Timeouts are 600s and each core uses 1.5 Gb RAM.

For each problem, we generate 3 sets of 15 instances. The first set consists of easy problems solveable by most of the solvers, the second set are of medium difficulty, and the third set are hard and cause many solvers to time out. We created new instances or modified the publicly available instances as necessary to satisfy these criteria. We have made available on

our website `http://people.eng.unimelb.edu.au/pstuckey/unsat_core`, high level models in MiniZinc [80] format, and MaxSAT/other generated models.

For each problem and instance set (easy, medium, or hard), each solver tackles each of the 15 instances 5 times with a different search order, based on randomly permuting the variables and constraints. We generate 5 permutations of each *FZN* (FlatZinc [11]) or *WCNF* (MaxSAT) or other model. The runtime of a solver on an instance is taken as the median of runtimes over the 5 permutations (with 600 s representing a timeout). For each solver we show the geometric mean of this runtime over the 15 instances in the set.

5.11.1 Resource Constrained Project Scheduling Problems with Weighted Earliness/Tardiness

The Resource Constrained Project Scheduling Problem (RCPSP) is, given a set of tasks, find a feasible schedule, as the start time of each task, within some predetermined planning horizon. The durations of the tasks are given, along with precedences between the tasks (e.g. prerequisite tasks which must be completed before a given task can be started), and a set of renewable resources such that the set of tasks scheduled at any given time must not together require more than the maximum available amount of any resource.

RCPSP with a Weighted Earliness/Tardiness objective (RCPSP/WET) [114] is, find the minimum cost schedule, given a deadline (ideal start time) for each task, and the costs per time unit deviation from the ideal schedule. Separate costs are given for earliness, if the task starts earlier than the deadline, or lateness, if the task starts later than the deadline. RCPSP/WET has previously been tackled with a CP-like `LSSU` approach with a very tight propagation strategy designed for RCPSP/WET, but without learning [114].

RCPSP/WET may be stated as follows.

$$
\begin{aligned}
&\text{find } x_1, \ldots, x_N, E_1, \ldots, E_N, T_1, \ldots, T_N \in 0..horizon \\
&\text{minimizing } \sum_{i=1}^{n} earliness\_cost_i E_i + \sum_{i=1}^{n} tardiness\_cost_i T_i \\
&\text{such that } E_i \geq deadline_i - x_i, T_i \geq x_i - deadline_i && \forall\, i \in 1..n, \\
&\qquad x_{succ\_task_i} - x_{pred\_task_i} \geq pred\_time_i && \forall\, i \in 1..P, \\
&\qquad cumulative([x_j]_{j=1}^{n}, [duration_j]_{j=1}^{n}, [resource\_usage_{i,j}]_{j=1}^{n}, resource\_limit_i) \\
&\qquad && \forall\, i \in 1..R
\end{aligned}
$$

constants n = number of tasks,

$horizon$ = maximum start time of any task,

$(earliness\_cost_i, tardiness\_cost_i, duration_i, deadline_i)$ = task i data,
$i \in 1..n$

P = number of precedences,

$(pred\_task_i, succ\_task_i, pred\_time_i)$ = precedence i data, $i \in 1..P$,

R = number of resources,

$([resource\_usage_{i,j}]_{j=1}^{n}, resource\_limit_i)$ = resource i data, $i \in 1..R$. (5.11)

The *cumulative* constraint used in this model is defined as follows.

$$cumulative([x_i]_{i=1}^{n}, [duration_i]_{i=1}^{n}, [resource\_usage_i]_{i=1}^{n}, resource\_limit) \equiv \bigwedge_{t=0}^{\max_{i=1}^{n} x_i} \left(\sum_{i \in 1..n, x_i \in t-duration_i+1..t} resource\_usage_i \leq resource\_limit \right) \quad (5.12)$$

To generate RCPSP/WET instances, we first generate RCPSP instances by selecting as many instances as necessary from PSPLib [63], specifically the *j30*, *j60* and *j90* sets which are of increasing difficulty level. Compared with Problem (5.11) above, the RCPSP instances have a *makespan* objective, i.e. minimizing the start time of a dummy task which completes after all others. There is no *deadline*, *earliness_cost*, or *tardiness_cost* in RCPSP.

Having selected 15 instances per set which can be solved as RCPSP, we take the start times in the RCPSP solution as a guide to choosing realistic deadlines for the RCPSP/WET problems we generate, because unsatisfiable-core solving is expected to be most useful when there is a reasonable expectation that soft constraints could be satisfied.

Given $x_1, \ldots, x_N$ from the RCPSP solution where x_N is the makespan, we take $slop = 0.2x_N$, and let the RCPSP/WET instance be the RCPSP instance with $horizon = x_N + slop$, and the additional RCPSP/WET constants chosen uniformly at random from $deadline_i \in x_i..x_i + slop$, $earliness\_cost_i \in 0..5$, $tardiness\_cost_i \in 0..5$ for all $i \in 1..n$.

In this experiment *LCG-glucose* uses the *cumulative* global propagators with explanations provided by Schutt et al. [102]. We try the basic `LSSU` solver with Schutt et al.'s timetable propagator (TT) or timetable-edge-finding propagator (TTEF). As we expected, the TT propagator performs better (TTEF is applicable to more highly-constrained problems), so all of the *LCG-glucose* unsatisfiable-core algorithms run with TT.

Table 5.1 shows the results of this experiment. The unsatisfiable-core algorithms `MaxRes` and `OLL` are clearly much better than the basic `LSSU` algorithm on this problem, with the remaining unsatisfiable-core algorithms `MSU1/3/4` performing similarly to `LSSU`. This is independent of the solver family (*LCG-glucose*, *WCNF-glucose*, or *WCNF-original*). Furthermore the LCG solvers are clearly much better than the MaxSAT solvers.

RCPSP/WET		j30		j60		j90	
algorithm	variant	TO	time s	TO	time s	TO	time s
LCG-glucose:							
WPM1		4	1.36	2	0.66	5	7.41
MaxRes		0	0.38	2	0.53	1	1.86
OLL	eager	0	0.28	1	0.45	**1**	**1.50**
	lazy	**0**	**0.26**	**1**	**0.44**	**1**	**1.50**
	LCG	0	0.30	1	0.51	1	1.74
MSU3		0	3.19	15	600.00	15	600.00
LSUS		0	5.60	15	600.00	15	600.00
MSU4		0	1.51	15	600.00	15	600.00
LSSU	TT	0	6.01	15	600.00	15	600.00
	TTEF	0	8.16	15	600.00	15	600.00
WCNF-glucose:							
WPM1		13	272.82	15	600.00	15	600.00
MaxRes		0	0.69	2	8.56	6	111.07
OLL	eager	3	7.04	4	73.00	10	226.37
	lazy	2	2.84	3	30.52	7	162.68
MSU3		11	513.60	15	600.00	15	600.00
LSUS		14	598.98	15	600.00	15	600.00
MSU4		12	552.70	15	600.00	15	600.00
LSSU		15	600.00	15	600.00	15	600.00
WCNF-original:							
WPM1	*MSUnCore*	14	583.98	15	600.00	15	600.00
MaxRes	*eva500a*	0	2.78	2	52.16	7	397.38
OLL	*MSCG*	0	0.61	2	6.06	4	58.72
MSU3	*MSUnCore*	6	155.52	15	600.00	15	600.00
MSU4	*MSUnCore*	6	195.04	15	600.00	15	600.00
LSSU	*clasp*	1	17.31	14	591.23	15	600.00
WCSP-original:							
LSSU	*toulbar2*	2	28.07	9	316.35	15	600.00
SMT2-original:							
MaxRes	*Z3-MaxSMT*	0	3.90	1	31.70	2	119.41

Table 5.1 Resource Constrained Project Scheduling Problems with Weighted Earliness/Tardiness experiment

Comparing *LCG-glucose* with *WCNF-glucose* shows that although the LCG solver is generating the same SAT problem as tackled by the MaxSAT solver, doing so lazily is a huge advantage. More difficult problems show a greater improvement. It is clearly worth using LCG with unsatisfiable-core solving (MaxRes or OLL) on this problem.

Comparing *WCNF-glucose* with *WCNF-original* shows that our algorithms are reasonably close to the publicly available ones. The differences concern the algorithms that use cardinality constraints (LSSU and MSU3/4, and to a lesser extent OLL). Different decompositions can be used, and also *MSCG* is described as reusing the decomposition literals of dominated cardinality constraints [77], which we omitted for simplicity. Since these prob-

lems go away in *LCG-glucose* which natively supports cardinality constraints, our relatively simplistic MaxSAT approach as described in Algorithm 5.1 is sufficient.

The extensional cost-shifting *WCSP* solver *toulbar2* was the best `LSSU` solver by a considerable margin, however it was not competitive with the unsatisfiable-core solvers. The `MaxRes`-based *SMT2* solver *Z3-MaxSMT* was somewhat better than the MaxSAT solvers, in particular *eva500a* which uses a similar backend (so SMT must be reducing decomposition overhead), but was not competitive with *LCG-glucose* with unsatisfiable cores.

5.11.2 Nurse Scheduling Problems

The Nurse Scheduling Problem (NSP) is, given a set of nurses, a planning horizon as the number of days and the number of shifts per day, and a set of nurse preferences as an integer cost per nurse, day and shift, find the minimum cost schedule, given shift pattern constraints such as the minimum rest period between shifts, number of consecutive shifts of a certain type, overall number of shifts allowed to be worked, etc.

NSP may be stated as follows.

$$\begin{aligned}
\text{find } & x_{1,1}, \ldots, x_{1,D}, \ldots, x_{N,1}, \ldots, x_{N,D} \in 1..S \\
\text{minimizing } & \sum_{i=1}^{N} \sum_{j=1}^{D} preference_{i,j,x_{i,j}} \\
\text{such that } & \sum_{k=1}^{D} (x_{i,k} = j) \in assignments_per_shift_min_j..assignments_per_shift_max_j \\
& \qquad \forall\, i \in 1..N, j \in 1..S, \\
& \sum_{k=1}^{N} (x_{k,i} = j) \geq coverage_{i,j} \qquad \forall\, i \in 1..D, j \in 1..S, \\
& regular(Q, q_0, next, accept, [x_{i,j}]_{j=1}^{N}) \qquad \forall\, i \in 1..n \\
\text{constants } & N = \text{number of nurses}, \\
& D = \text{number of days}, \\
& S = \text{number of shifts +1 such that shift } S \text{ means rest}, \\
& preference_{i,j,k} = \text{nurse } i \text{ day } j \text{ shift } k \text{ cost}, \quad i \in 1..N, j \in 1..D, k \in 1..S, \\
& (assignments_per_shift_min_i, assignments_per_shift_max_i) = \\
& \qquad \text{pattern rule data}, \qquad i \in 1..s, \\
& coverage_{i,j} = \text{number of nurses required}, \quad i \in 1..D, j \in 1..S \\
& (Q, q_0, next, accept) = \text{pattern rule Discrete Finite Automaton (DFA)}.
\end{aligned}$$

NSP algorithm	variant	14_10 TO	14_10 time s	21_15 TO	21_15 time s	28_20 TO	28_20 time s
LCG-glucose:							
WPM1		8	18.69	11	85.61	11	154.43
MaxRes		6	11.23	8	42.81	10	82.86
OLL	eager	3	4.72	6	32.24	10	76.66
	lazy	2	4.04	6	32.35	10	74.86
	LCG	**1**	**3.44**	**5**	**24.48**	**10**	**69.18**
MSU3		10	43.11	13	294.89	14	364.68
LSUS		10	49.22	13	306.62	14	366.39
MSU4		10	42.47	13	314.08	14	364.60
LSSU		10	47.96	13	301.06	14	372.84
WCNF-glucose:							
WPM1		8	18.02	11	91.79	11	158.52
MaxRes		5	6.60	9	47.57	10	94.14
OLL	eager	3	3.64	9	44.55	10	95.80
	lazy	2	2.97	9	39.34	10	88.59
MSU3		10	57.42	13	364.04	14	465.04
LSUS		10	174.20	15	600.00	15	600.00
MSU4		10	44.94	13	325.28	14	379.52
LSSU		10	63.60	13	377.97	15	600.00
WCNF-original:							
WPM1	*MSUnCore*	8	43.96	11	173.77	11	300.57
MaxRes	*eva500a*	5	22.78	9	118.83	10	275.83
OLL	*MSCG*	5	7.06	9	53.95	10	106.24
MSU3	*MSUnCore*	8	40.73	10	227.08	14	463.22
MSU4	*MSUnCore*	8	58.27	10	281.39	14	498.28
LSSU	*clasp*	10	78.22	14	462.84	14	546.38
WCSP-original:							
LSSU	*toulbar2*	11	304.06	15	600.00	15	600.00
SMT2-original:							
MaxRes	*Z3-MaxSMT*	4	27.48	8	241.18	10	503.10

Table 5.2 Nurse scheduling experiment

The *regular* constraint used in this model is defined as follows.

$$regular(Q, q_0, next, accept, [x_i]_{i=1}^{\ell}) \equiv \exists state_0, \ldots, state_\ell \text{ such that}$$
$$state_0 = q_0 \wedge state_\ell \in accept \wedge \bigwedge_{i=1}^{\ell} ((state_{i-1}, state_i, x_i) \in next) \tag{5.13}$$

To generate NSP instances, we start with a 28-day 30-nurse instance from the $N30$ data-set of NSPLib, and the pattern rules from case 9 of NSPLib [113]. We reduce D by truncating preference and coverage data, and scaling pattern rules such as the min/max number of assignments per shift (e.g. by 1/2 for a 14-day problem). We reduce N by truncating preference data and scaling coverage data (e.g. by 1/3 for a 10-nurse problem).

We select 15 feasible instances from NSPLib (defined as instances where *LCG-glucose* finds any solution to the 28-day 20-nurse problem within one minute). We generate the easy, medium and hard instance sets from these 15 instances. We name the instance sets *D_N* i.e. the number of days followed by the number of nurses.

In this experiment *LCG-glucose* implements the *regular* constraint using the Multi-valued Decision Diagram (MDD) global propagator with explanations from Gange et al. [49]. We generate an MDD by transforming the DFA $(Q, q_0, next, accept)$ into a layer-graph as an automaton of $d \times Q$ states, then removing unreachable states and merging indistinguishable states. This yields a compact description of the allowable shift pattern.

The remaining solvers are the MaxSAT solvers and the additional solvers *toulbar2* [96] and *Z3-MaxSMT* [18]. For these solvers we decompose the *mdd* (optimized *regular*) constraint similarly to definition (5.13), but we take *next* as a function $state_{i-1} \times x_i \rightarrow state_i$, implemented as a 2-dimensional constant array or lookup table.

Table 5.2 shows the results of this experiment. The unsatisfiable-core algorithms `MaxRes` and `OLL`, and to a lesser extent `MSU1`, are clearly much better than the basic `LSSU` algorithm on this problem, with the remaining unsatisfiable-core algorithms `MSU3/4` performing similarly to `LSSU`. The LCG solvers were somewhat better than the MaxSAT solvers.

In this case the improvement due to LCG was not as marked as in the RCPSP/WET experiment, and also wasn't as marked on the difficult instances compared with easy and medium. This occurs because (i) the SAT decomposition, whilst having greater propagation overhead and requiring decomposition literals as compared with the *mdd* propagator, does produce more compact explanations and so we expect it to be competitive, and (ii) the advantage of an integer linear objective was not as marked compared with RCPSP/WET, since objective terms (preferences) are small in value, having domain 1..4 only.

Interestingly, and unlike the RCPSP/WET experiment, our simplistic MaxSAT approach, as in Algorithm 5.1, was better than the publicly available MaxSAT solvers. The extensional cost-shifting *WCSP* solver *toulbar2* was somewhat competitive with the other `LSSU` solvers, but was not the best performing `LSSU` solver in this experiment. Again, *Z3-MaxSMT* was better than the MaxSAT solvers, but not competitive with *LCG-glucose*.

5.11.3 Soft Car Sequencing

The Soft Car Sequencing problem [105] is, given a set of car types and options that must be fitted to each car, find a minimum-cost schedule as a permutation of the cars to produce, minimizing overloads of the machines on the production line which fit each kind of option. If N cars must be produced and a particular option can be fitted to u of every w cars that go past, then a sliding window encompassing w consecutive cars in the schedule must be examined starting at every position $1..N - w + 1$ of the schedule. Any position where more than u cars in that window require the option incurs a one-unit penalty.

Soft Car Sequencing may be stated as follows.

$$\begin{aligned}
&\text{find } x_1, \ldots, x_N \in 1..T \\
&\text{minimizing } \sum_{i=1}^{O} \sum_{j=1}^{N-w_i+1} \left(\sum_{k=j}^{j+w_i-1} (x_k \in \mathit{option_to_types}_i) > u_i \right) \\
&\text{such that } \sum_{j=1}^{N} (x_j = i) = \mathit{quantity}_i && \forall i \in 1..T \\
&\text{constants } N = \text{number of cars to produce,} \\
&\qquad T = \text{number of car types,} \\
&\qquad O = \text{number of options,} \\
&\qquad u_i = \text{upper limit of option } i, && i \in 1..O, \\
&\qquad w_i = \text{window size of option } i, && i \in 1..O, \\
&\qquad \mathit{option_to_types}_i = \text{set of car types requiring option } i, && i \in 1..O, \\
&\qquad \mathit{quantity}_i = \text{number of cars of type } i \text{ to produce,} && i \in 1..T.
\end{aligned}$$

To generate Soft Car Sequencing instances, we start with a 100-car or 200-car 5-option problem from CSPLib [53]. The number of car types ranges from 17 to 30 in these problems. Then given a revised number of cars N to produce (40, 50 or 60 in this experiment), we generate a new *quantity* vector, which sums to N, and uses as many options as possible such that the utilization of any option does not exceed 90%. The new problems use the original *option_to_types* data, but don't necessarily use all of the car types. We name the instance sets *N_90* i.e. the number of cars followed by the utilization percentage.

Although the model contains a Global Cardinality Constraint (GCC) based on the *quantity* vector, the *gcc* global propagator [91] wasn't useful in these experiments. All solvers decode options into car types by a SAT decomposition, so any advantage for LCG solving is by supporting cardinality constraints directly instead of through decomposition. The MaxSAT solvers use the decomposition of Abío et al. [1] for cardinality as usual.

Table 5.3 shows the results of this experiment. The unsatisfiable-core algorithms `MaxRes` and `OLL`, and to a lesser extent `MSU1`, are clearly much better than the basic `LSSU` algorithm on this problem, with the remaining unsatisfiable-core algorithms `MSU3/4` performing similarly to `LSSU`. The LCG solvers were somewhat better than the MaxSAT solvers.

Our version of `MaxRes` was better than the publicly available version. Interestingly, *LCG-glucose*'s native support for the cardinality constraints generated by `OLL` wasn't useful in this experiment. The best unsatisfiable-core algorithm is `MaxRes`, probably due to its low overhead when there are few violations and few overlapping unsatisfiable cores.

The extra solvers *toulbar2* and *Z3-MaxSMT* were not useful at all on this model. In the case of *toulbar2* this may be because the unavoidable decomposition of linear constraints to

car_sequence		40_90		50_90		60_90	
algorithm	variant	TO	time s	TO	time s	TO	time s
LCG-glucose:							
WPM1		0	1.35	5	32.32	10	190.08
MaxRes		0	1.24	**3**	**25.53**	**7**	**151.81**
OLL	eager	1	10.77	8	219.46	14	499.76
	lazy	**0**	**1.20**	4	31.00	9	168.74
	LCG	0	2.15	6	62.39	11	262.36
MSU3		0	2.23	5	61.27	11	273.57
LSUS		0	2.31	5	59.97	11	269.40
MSU4		0	2.36	7	64.01	11	280.16
LSSU		0	2.32	5	65.38	11	269.58
WCNF-glucose:							
WPM1		0	1.78	7	51.01	12	264.11
MaxRes		0	1.50	5	44.41	11	250.42
OLL	eager	0	7.94	9	137.01	13	472.40
	lazy	0	1.32	5	45.11	11	241.13
MSU3		0	1.41	5	45.10	11	254.92
LSUS		0	1.46	5	45.83	12	263.67
MSU4		0	4.08	9	172.81	13	395.58
LSSU		0	14.16	8	292.53	13	526.85
WCNF-original:							
WPM1	*MSUnCore*	0	1.97	5	59.48	11	264.96
MaxRes	*eva500a*	0	7.38	7	128.15	12	364.45
OLL	*MSCG*	0	1.42	5	41.70	10	255.03
MSU3	*MSUnCore*	0	1.78	5	64.51	11	283.97
MSU4	*MSUnCore*	0	21.55	6	299.63	13	546.36
LSSU	*clasp*	0	3.26	5	54.54	11	246.56
WCSP-original:							
LSSU	*toulbar2*	15	600.00	15	600.00	15	600.00
SMT2-original:							
MaxRes	*Z3-MaxSMT*	13	489.39	15	600.00	15	600.00

Table 5.3 Soft Car Sequencing experiment

binary operations is not efficient. In the case of *Z3-MaxSMT* this may be due to our integer representation of $x_1, \ldots, x_N$ where a one-hot encoding [17] might be better. Based on the previous experiments, we felt it was not worth pursuing this in detail.

5.11.4 Curriculum Timetabling

The Curriculum Timetabling (CTT) problem [37] is, given a set of courses each taught by a given teacher and consisting of a given number of lectures, schedule the lectures over a given number of rooms and periods, minimizing hard constraint violations such clashes between courses taught by the same teacher or that can be studied simultaneously (as defined by *curricula* or sets of courses), and soft constraint violations such as those concerning room

capacities or spare periods or days (given the number of periods per day). CTT has previously been tackled with SAT/MaxSAT [9] despite its large SAT decomposition.

CTT may be stated as follows.

$$\begin{aligned}
&\text{find } x_{1,1}, \ldots, x_{1,P}, \ldots, x_{C,1}, \ldots, x_{C,P} \in 0..R \\
&\text{minimizing } (z_h, z_l) \text{ by lexical comparison; } z_h \text{ is primary objective, } z_l \text{ secondary} \\
&\text{where } z_h = \mathit{lecture_violation} + \mathit{room_occupancy_violation} + \\
&\qquad \mathit{conflict_violation} + \mathit{availability_violation}, \\
&\qquad z_l = \mathit{room_capacity_cost} + \mathit{min_working_days_cost} + \\
&\qquad \mathit{curriculum_compactness_cost} + \mathit{room_stability_cost},
\end{aligned}$$

where $x_{i,j} = 0$ if no lecture is scheduled for course i in period j, or $x_{i,j} \in 1..R$ if a lecture is scheduled in the given room. The hard constraints, or primary objective, are defined by

$$\begin{aligned}
\mathit{lecture_violation} &= \sum_{i=1}^{C} \left(\mathit{course_lectures}_i - \sum_{j=1}^{P} (x_{i,j} \neq 0) \right), \\
\mathit{room_occupancy_violation} &= \sum_{i=1}^{R} \sum_{j=1}^{P} (\max(1, \mathit{room_period}_{i,j}) - 1), \\
\mathit{conflict_violation} &= \sum_{(i,j) \in \mathit{conflicts}} \sum_{k=1}^{P} (x_{i,k} \neq 0)(x_{j,k} \neq 0), \\
\mathit{availability_violation} &= \sum_{(i,j) \in \mathit{unavailabilities}} (x_{i,j} \neq 0)
\end{aligned}$$

and the soft constraints, or secondary objective, are defined by

$$\begin{aligned}
\mathit{room_capacity_cost} &= \sum_{i=1}^{C} \sum_{j=1}^{R} \max(0, \mathit{room_capacity}_j - \mathit{course_students}_i) \mathit{course_room}_{i,j}, \\
\mathit{min_working_days_cost} &= 5 \sum_{i=1}^{C} \max \left(0, \mathit{min_working_days}_i - \sum_{j=1}^{D} \mathit{course_day}_{i,j} \right), \\
\mathit{curriculum_compactness_cost} &= 2 \sum_{i=1}^{A} \sum_{j=1}^{D} \sum_{k \in \mathit{day_to_periods}_j} \\
&\qquad \mathit{curriculum_period}_{i,k} \bigwedge_{\ell \in \{k-1, k+1\} \cap \mathit{day_to_periods}_j} (\mathit{curriculum_period}_{i,\ell} = 0), \\
\mathit{room_stability_cost} &= \sum_{i=1}^{C} \left(\max \left(1, \sum_{j=1}^{R} \mathit{course_room}_{i,j} \right) - 1 \right),
\end{aligned}$$

based on the convenience functions

$$curriculum\_period_{i,j} = \sum_{k \in curriculum\_courses_i} (x_{k,j} \neq 0) \qquad \forall i \in 1..A, j \in 1..P,$$

$$course\_day_{i,j} = \bigvee_{k \in day\_to\_periods_j} (x_{i,k} \neq 0) \qquad \forall i \in 1..C, j \in 1..D,$$

$$course\_room_{i,j} = \sum_{k=1}^{P} (x_{i,k} = j) \qquad \forall i \in 1..C, j \in 1..R,$$

$$room\_period_{i,j} = \sum_{k=1}^{C} (x_{k,j} = i) \qquad \forall i \in 1..R, j \in 1..P,$$

and the problem specification, or instance data,

constants C = number of courses,

$(course\_lectures_i, course\_students_i)$ = course i data, $i \in 1..C$,

P = number of periods,

D = number of days in P periods,

$day\_to\_periods_i$ = set of periods in day i, $i \in 1..D$,

R = number of rooms,

$room\_capacity_i$ = maximum number of students in room i, $i \in 1..R$,

A = number of curricula,

$curriculum\_courses_i$ = set of courses in curriculum i, $i \in 1..A$,

conflicts = set of pairs (i, j) of conflicting courses i and j where $i < j$,

unavailabilities = set of pairs (i, j) where course i can't be in period j.

To generate Curriculum Timetabling instances, we use Lopes and Smith-Miles's problem generator [64], with $events = 50, 100, 150$ and $occupancy = 30\%, 50\%, 70\%$ for the easy, medium and hard instance sets respectively. This generator takes a graph-theoretic approach to generating conflicts and the resulting instances are challenging, indeed some instances require $z_h > 0$, which means that the conflicts are quite fundamental.

Although the model contains overlapping Global Cardinality Constraints (GCC) defining the *course_room* and *room_period* intermediate variables, and indeed *conflict_violation* defines a soft *alldifferent_except_0* constraint, global *gcc*/*alldifferent* propagators [90, 91] weren't useful on this problem. We implemented the model by clausal, *linear*, and *max* constraints, plus a linear objective. For LCG all constraints are natively supported. For MaxSAT we decompose *linear* constraints per Abío et al. [1], and *max* constraints with an upfront decomposition using the same clauses generated lazily in LCG.

CTT		50_30		100_50		150_70	
algorithm	variant	TO	time s	TO	time s	TO	time s
LCG-glucose:							
WPM1		0	0.65	3	11.51	5	58.84
MaxRes		**0**	**0.64**	3	11.36	6	57.40
OLL	eager	0	0.65	3	11.41	6	59.16
	lazy	0	0.67	**3**	**10.88**	6	60.03
	LCG	0	0.65	3	11.55	**4**	**58.65**
MSU3		0	0.87	4	18.76	7	64.99
LSUS		0	0.92	4	22.83	7	69.57
MSU4		0	0.90	4	18.82	7	64.29
LSSU		2	56.66	12	470.96	15	600.00
WCNF-glucose:							
WPM1		0	0.96	4	30.96	8	269.70
MaxRes		0	1.09	3	35.48	8	265.41
OLL	eager	1	1.74	6	55.55	8	278.61
	lazy	0	1.07	3	35.75	8	271.92
MSU3		0	1.62	4	53.64	8	260.33
LSUS		0	1.73	6	65.29	9	277.42
MSU4		1	2.55	7	68.20	8	280.73
LSSU		15	600.00	15	600.00	15	600.00
WCNF-original:							
WPM1	*MSUnCore*	0	3.43	3	69.84	9	273.99
MaxRes	*eva500a*	0	32.21	7	366.67	15	600.00
OLL	*MSCG*	0	2.70	3	42.80	6	250.26
MSU3	*MSUnCore*	0	7.65	4	121.82	10	377.46
MSU4	*MSUnCore*	3	15.86	9	251.96	10	385.25
LSSU	*clasp*	0	31.62	6	423.24	14	597.99

Table 5.4 Curriculum Timetabling experiment

This model is lengthy, uses the unusual constraint *max*, and requires clever encoding to reduce the quadratic overhead of pairwise constraints like *curriculum_compactness_cost*. So we did not attempt to map this model to the other solvers *toulbar2* and *Z3-MaxSMT*, given the previous experiments had shown these solvers can't compete with *LCG-glucose*. One interesting feature of *Z3-MaxSMT* is that it natively handles a lexical objective (z_h, z_l) as occurs in this model, but took a simpler approach of optimizing $1000z_h + z_l$, which worked well with all unsatisfiable-core algorithms including our generalizations to LCG.

Table 5.4 shows the results of this experiment. The LCG solvers are clearly much better than the MaxSAT solvers. The quadratic numbers of most constraints in this problem (duplicated per course and period, or room and period, etc), leads to a very large MaxSAT decomposition which is difficult for the MaxSAT solvers. This holds independently of solving algorithm. The unsatisfiable-core algorithms performed similarly to each other, and are clearly much better than the basic LSSU algorithm. This holds independently of solver. The best unsatisfiable-core algorithm is OLL using LCG for cardinality constraints.

Our simplified unsatisfiable-core MaxSAT solvers are also significantly better than the publicly available solvers, in particular *eva500a* which seems to use unsatisfiable-core minimization [13]. An exception is that *WCNF-glucose* using the *LSSU* algorithm was worse than *clasp*, since the former did not solve any instances. This may be due to (i) *clasp* using LCG techniques internally, whereas *WCNF-glucose* relies on decomposition, or (ii) *clasp* using a phase selection strategy that minimizes the objective, whereas *WCNF-glucose* uses *glucose*'s default strategy. We omitted such techniques for simplicity. In any case, the unsatisfiable-core algorithms greatly prefer *glucose*'s default phase-selection strategy.

5.12 Our early unsatisfiable-core work

The unsatisfiable-core experiments, whilst giving promising results in their current form, were very disappointing initially. This was due to a 'perfect storm' of reasons.

Initially, we did not have good insight into `MSU1/WPM1` since they were tightly integrated into the SAT solver (use of resolution tracing, clause rewriting, clause splitting, etc), and so apart from trying a more-or-less direct embedding of `MSU1` into LCG[3], we mainly focused on `MSU3/4` as better candidates for adaptation to integer problems.

We noticed that on competition instances [8], the `MSU1/WPM1` solvers either solved instances instantly (in less than 0.1s) or did not solve them at all. This seemed at odds with their very good scores in the competition overall. We reasoned that this was due to symmetries, thus independently discovering Ansótegui et al.'s observation [7]. Although we had some complicated ideas to remove the symmetries, such as using Network Flow to assign violations, we put this aside and continued to focus on `MSU3/4`.

We made significant efforts to improve `MSU3/4`, defining several alternative versions of the algorithm, which did very well on specific problems[4]. But to our frustration, we could never make our `MSU3/4` variants competitive on problems that `MSU1/WPM1` solves well. The MaxSAT decomposition of a large problem would often be solved faster by *MSUnCore* with `MSU1/WPM1` than the LCG version using *Chuffed* [22] with our algorithms.

Eventually we decided to focus more closely on `MSU1/WPM1`. We defined several integer versions of the algorithms, and gradually started to decouple the ideas of the algorithms from their tightly integrated SAT implementation. We implemented alternative integer extensions to `MSU1/WPM1` in an early *CPX* [45] prototype, but this didn't result in a paper.

This gave a more-or-less workable platform for experiments that was potentially competitive with `MSU1/WPM1`, but the remaining challenge (not entirely clear to us at the time, since there were so many problems to work through), was the significant differences between our basic solver *CPX* as compared with *MSUnCore*'s basic solver *PicoSAT* [16].

[3] Downing, N., Feydy, T., Stuckey, P.J.: Unsatisfiable cores for constraint programming. *CoRR abs/1305.1690* (2013). URL `http://arxiv.org/abs/1305.1690`

[4] Downing, N., Feydy, T., Stuckey, P.J.: Unsatisfiable cores and lower bounding for constraint programming. *CoRR abs/1508.06096* (2015). URL `http://arxiv.org/abs/1508.06096`

5.13 Our new *LCG-glucose* solver

We were certain that if a problem uses integer variables and global constraints, then adding an integer variable model and global constraint propagators by adding LCG to MaxSAT, could only improve performance as compared with running the decomposition on a basic MaxSAT solver. Unfortunately, the results seemed to say otherwise.

These poor results occurred with *CPX*, because of *CPX*'s weaker activity-based search which is integer-variable rather than literal based, and *CPX*'s much slower semantic conflict analysis and generally slow performance on SAT-like problems. *CPX* is optimized for difficult CP problems making heavy use of integer variables with huge domains.

We reasoned that since MaxSAT solvers are highly optimized for the very simple problem type that they solve, we would have to design an incredibly fast LCG solver to show the benefits of LCG propagation over the equivalent but somewhat larger decompositions tackled by MaxSAT. In addition, by designing our own solvers we could carefully control the other factors such as the activity-based search, to make a proper comparison.

The new solvers *WCNF-glucose* and *LCG-glucose* took 15 months to develop, probably 7.5 months full time, as compared with the similar amounts of time spent learning each of the *Chuffed* and *CPX* codebases to the point where we could fix bugs in the solver core, etc (we are certainly indebted to the authors of those solvers for the insights gained).

We submitted a brief description[5] of *LCG-glucose* to the MiniZinc Challenge 2016. The major innovations are the watch lists and wakeups, which are similar to a SAT solver but extended to the LCG integer variable model and LCG propagators with very little overhead, and the minimal set of highly-incremental constraint propagators.

Another important feature is an efficient implementation of the half-reification transformation [44] in which a constraint like $a \leftrightarrow C$ is decomposed to $a \rightarrow C \wedge a \leftarrow C$ and then further optimized based on how a apppears elsewhere, and an extension of the same approach to *linear* and related constraints in which a constraint like $x = f(y_1, \ldots, y_n)$ is decomposed to $x \leq f(\ldots) \wedge x \geq f(\ldots)$ and then further optimized.

The solvers are highly modular, comprising a frontend, a model transformation layer, and a backend. The model transformation layer presents a simple and customizable interface to frontend and backend, making it easy to write a new frontend or backend.

We did this so that the LCG backend used in *LCG-glucose*, the unmodified *glucose* backend used in *WCNF-glucose*, and the unmodified MaxSAT solvers could all work on appropriately decomposed versions of the exact same model, utilizing all transformations and avoiding any bias. We planned to implement *toulbar2* and *Z3* backends if time.

[5]Downing, N.: MiniZinc challenge 2016: LCG-glucose solver description. Tech. rep., Data61 CSIRO; University of Melbourne, Australia (2016). URL `http://www.minizinc.org/challenge2016/description_lcg-glucose.txt`

5.13.1 Highly-incremental *linear* propagator in *LCG-glucose*

LCG-glucose uses a highly-incremental *linear* propagator, which is extremely simple and follows a similar approach to *clasp*'s Pseudo-Boolean propagator [51] in maintaining the slack in the constraint by a very fast wakeup on forward propagation or backtracking. This led to the interesting issue that we could not duplicate some previous global propagator results[6], since *linear* decompositions become extremely competitive.

5.13.2 Highly-incremental *cumulative* propagator in *LCG-glucose*

One final noteworthy feature of *LCG-glucose* is our highly-incremental timetable-based *cumulative* propagator, based on Schutt et al.'s [99] which is not incremental. The *cumulative* constraint (5.12), used in Section 5.11.1, states that given a set of tasks with a start time and duration, the set of tasks running at any time does not use more than some given amount of a given resource. The *timetable* is an internal structure which states how much of the given resource is in use at any given time, based on tasks known to be running.

We implement the timetable as a dictionary $TT = \{t_1 \mapsto u_1, t_2 \mapsto u_2, \ldots, t_n \mapsto u_n\}$ mapping time t_i to usage u_i for $i \in 1..n$. Each 'break' $i \in 1..n-1$ states that the usage is u_i at time t where $t_i \leq t < t_{i+1}$. We set $t_1 = 0$ and $u_n = 0$. Given task j starts at time x_j and has duration $duration_j$, it must execute between its $latest\_start\_time = \mathrm{ub}(x_j)$ and $earliest\_finish\_time = \mathrm{lb}(x_j) + duration_j$. As $D(x_j)$ shrinks, the incremental propagator wakes up and adds usage to the timetable, removing it again on backtracking.

We detect failure of the *cumulative* constraint directly in the wakeup (timetable update) routine. We also keep track of the minimal slack in the timetable during updates, and schedule a conventional propagator execution when it appears this might be small enough to cause propagation. The propagator is conventional yet fast, because it uses the existing timetable which is always up to date. We use a more compact explanation than Schutt et al.'s because we define 'task j runs at time t' literals, which are the same as in MiniZinc's standard library [11] *cumulative* decomposition, but lazily created as required.

However, in the end we decided not use the new incremental *cumulative* propagator in this chapter's experiments, because the non-incremental propagator [99] was perfectly good enough once we had solved all the other problems, and served to illustrate the advantage of LCG over a MaxSAT decomposition. We prefer to write an 'engineering' paper in the future along the lines of Feydy and Stuckey [41], in which we plan to evaluate separately each proposed improvement of *LCG-glucose* over previous methods.

[6]Downing, N., Feydy, T., Stuckey, P.J.: Explaining flow-based propagation. In: N. Beldiceanu, N. Jussien, É. Pinson (eds.) *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems: 9th International Conference, CPAIOR 2012, Nantes, France, May 28 – June 1, 2012. Proceedings*, pp. 146–162. Springer Berlin Heidelberg (2012)

5.14 Conclusions

We achieved our aim of generalizing unsatisfiable-core algorithms from MaxSAT to Constraint Programming (CP), using a Lazy Clause Generation (LCG) solver to generate unsatisfiable cores. This required us to analyze the existing algorithms to determine the common features and bring implementation choices into conformity, and then define an efficient way to handle arbitrary integer linear terms in the optimization objective.

We showed that there exist suitable CP problems, in which many integer terms in the objective are either zero or close to zero, such that aggressively assuming them to be zero is beneficial. Using an LCG approach with unsatisfiable-core solving, we produced what we believe are state-of-the-art solutions to RCPSP/WET, and highly competitive solutions to a number of other standard problems which we solve to optimality.

The best unsatisfiable-core algorithm was our generalization of `OLL` [5, 77]. The `OLL` algorithms, including our generalization, fit more naturally into an LCG than a MaxSAT framework, since they have a concise definition using *linear* constraints, which are natively supported by LCG. Our generalization of `MaxRes` [79] was also very competitive.

No modelling effort was required in these experiments, since our unsatisfiable-core CP solvers can read a suitable optimization model directly from a FlatZinc [11] file. Our results clearly show that CP solvers using LCG, which can generate unsatisfiable cores, should offer an unsatisfiable-core optimization mode for use on suitable problems.

Chapter 6

Conclusions

6.1 Results – *alldifferent* constraints

In Chapter 3 we implemented the most important *alldifferent* propagators used in traditional CP, extended to explain themselves by clauses, and also *alldifferent* by decomposition. We compared against *Gecode* 4.4.0 [52], a state-of-the-art CP solver without learning.

The experiments demonstrate that LCG changes the tradeoffs for propagation. Whereas expensive propagation algorithms can pay for themselves in CP where the reduction in search is considerable, they have more difficulty paying for themselves in LCG.

The experiments also demonstrate that LCG changes the tradeoffs for decomposition. Whereas decomposition is usually a big disadvantage in CP where the lost globality through decomposition leads to extra search, it is less of a disadvantage in LCG.

Taken together, the above results show that LCG can discover the stronger consequences of *alldifferent* through learning, and thereby wholly or partially recover the globality lost through using a cheaper propagator for *alldifferent* or indeed a decomposition.

Overall, the best *alldifferent* propagation algorithm was the bounds-consistent propagator with explanations, as it is relatively cheap to execute and produces very compact explanations. The next best approach was the Feydy-consistent decomposition, which has slightly stronger propagation in some situations, and propagates efficiently if not too large.

LCG solvers should still offer a range of propagation approaches to *alldifferent*, since some problems benefit from dedicated propagators even in LCG. In such cases, e.g. Superblk [68], the search reduction due to higher consistency and the search reduction due to learning can be orthogonal, leading to huge benefits using both together.

6.2 Results – network flow constraints

In Chapter 4 we added the *min_cost_flow* propagator used in traditional CP (called *network-flow* in *JaCoP*), and a version without costs called *circulation* which can also implement *alldifferent* and related constraints. We compared against *PaLM* [60], an earlier CP solver with learning, and *JaCoP* 4.4.0 [106], a state-of-the-art CP solver without learning.

The experiments show that in an LCG solver, a generic flow-based propagator can be useful to propagate constraints such as *alldifferent* which are expressible as flow, and that the use of 1UIP nogoods [73] as opposed to decision nogoods [36] is essential.

The experiments demonstrate that, as in Chapter 3, LCG changes the tradeoffs for propagation. Whereas expensive propagation algorithms, such as the explicit arc-bounds-testing method implemented by *JaCoP*, can pay for themselves in CP where the reduction in search is considerable, they have more difficulty paying for themselves in LCG.

For problems explicitly using flow, the experiments show that with or without learning, a dedicated flow-based propagator is absolutely essential, since decomposition to primitive constraints is not useful on these problems, and that the dedicated flow-based propagator with learning is usually vastly superior to the same propagator without learning.

Therefore LCG solvers should definitely offer dedicated flow-based constraints with explanation capability. The standard SCC-based propagation method (extended to explain itself) is cheap and helpful on problems whose flow network has the right structure, whereas our new cut-cache (which is only applicable to LCG solvers) is helpful for problems with costs, since it can often avoid running the propagator which is quite expensive.

6.3 Results – unsatisfiable-core optimization

In Chapter 5 we generalized the `MSU1/WPM1`, `MaxRes`, `OLL`, and `MSU3/4` unsatisfiable-core optimization algorithms from MaxSAT to CP, using LCG to generate the unsatisfiable cores. We made a comprehensive analysis of the similarities and differences between the existing algorithms, in order to do this consistently for all algorithms considered.

Using a suite of scheduling and rostering problems, we compared our LCG solver with unsatisfiable-core optimization, running a native CP model of each problem, against our own MaxSAT solver, the MaxSAT solvers *MSUnCore*, *eva500a*, *MSCG*, *clasp*, and other solvers *toulbar2*, *Z3*, running an appropriate decomposition of each problem.

We showed that there exist suitable CP problems, in which many integer terms in the objective are either zero or close to zero, such that aggressively assuming them to be zero is beneficial. We produced what we believe are state-of-the-art solutions to RCPSP/WET, and highly competitive solutions to the other problems that we considered.

The best unsatisfiable-core algorithm was our generalization of `OLL` [5, 77]. The `OLL` algorithms, including our generalization, fit more naturally into an LCG than a MaxSAT

framework, since they have a concise definition using *linear* constraints, which are natively supported by LCG. Our generalization of `MaxRes` [79] was also very competitive.

No modelling effort was required in these experiments, since our unsatisfiable-core CP solvers can read a suitable optimization model directly from a FlatZinc [11] file. Our results clearly show that CP solvers using LCG, which can generate unsatisfiable cores, should offer an unsatisfiable-core optimization mode for use on suitable problems.

6.4 Discussion

The experiments in Chapters 3 and 4 show that LCG changes the tradeoffs for propagation. Whereas expensive propagation algorithms with stronger filtering, such as domain-consistent *alldifferent*, can pay for their propagation cost in CP where the reduction in search is considerable, they do not always pay for themselves in LCG.

For certain classes of problems a more expensive propagator does pay for itself, even in LCG which does well without such propagators. The bounds-consistent *alldifferent* propagator was useful, but mainly because it is very cheap to execute relative to its propagation strength. For *circulation* and *min_cost_flow* the SCC-based propagation algorithm usually helped, especially on 0-1 problems like *alldifferent* flow networks. Our novel cut-cache for network flows also gave a modest but consistent improvement.

The experiments in Chapters 3 and 4 also show that LCG changes the tradeoffs for decomposition. Whereas decomposition is often a big disadvantage in CP, where the lost globality through decomposition leads to extra search, it is less of a disadvantage in LCG. Indeed decomposition can introduce extra variables that improve learning.

For *alldifferent* a surprise good performer was the Feydy and Stuckey decomposition, showing that LCG can recover lost globality through learning. On the other hand, global *circulation* and *min_cost_flow* constraints encode much structure, and decomposition to *linear* constraints lost too much globality to be competitive even in LCG.

The experiments in Chapter 5 show that a dual approach to optimization in CP, in which we start from a highly ideal but infeasible solution and work to make this solution feasible, can be highly beneficial as compared with the traditional approach, in which we start from a feasible but non-ideal solution and work to make this solution more ideal.

As well as the unsatisfiable-core LCG solver being much faster than the ordinary LCG solver on the right problems, it was also much faster than the best unsatisfiable-core MaxSAT solvers on the same problems. These results improved with problem size.

This was further reinforced by the results of the 2016 MiniZinc Challenge [30]. We submitted our unsatisfiable-core LCG solver resulting from this research, to participate in the free search category. On 6 of the 17 optimization problems in the challenge, our unsatisfiable-core LCG solver was faster than our ordinary LCG solver.

6.5 Future directions

As to our research question, can LCG tackle large, realistic scheduling and rostering problems, the answer is absolutely *yes* – but it would be better if the core LCG solver had better support for these kinds of problems in several areas, which we discuss next. We then propose some further applications of LCG, and some extensions to different kinds of problems.

6.5.1 Improvements to LCG – the *linear* explanations

While the *linear* propagator [83], discussed in Section 2.5.1, is very efficient in the forward direction (propagation), the explanations that it produces in the backward direction (conflict analysis) are not ideal. This should be improved, as we will explain.

Reducing a violated *linear* constraint to a violated clause, or equivalently a *linear* constraint that propagates to a clause that propagates, requires us to fix all variables in the constraint, producing a clause which is much weaker than the original *linear* constraint. That is, the clause and subsequent clauses learnt from it, are not very reuseable.

Example 6.1 Consider $x_1 + \ldots + x_5 \geq 10$ where $x_1, \ldots, x_5 \in 1..10$. Suppose in the current subproblem $x_1, \ldots, x_4 \leq 2$. To propagate x_5 the solver checks the equivalent constraint

$$x_5 \geq 10 - x_1 - x_2 - x_3 - x_4 \tag{6.1}$$

whose RHS is at least $10 - 2 - 2 - 2 - 2 = 2$. Therefore it sets $x_5 \geq 2$ with explanation

$$[x_1 \leq 2] \wedge [x_2 \leq 2] \wedge [x_3 \leq 2] \wedge [x_4 \leq 2] \rightarrow [x_5 \geq 2]. \tag{6.2}$$

Now consider a similar subproblem where $x_1, x_2 \leq 3$ and $x_3, x_4 \leq 1$. The parent constraint (6.1) detects that $x_5 \geq 2$ exactly as before, but the clause (6.2) does not propagate, because the preconditions, in particular $[x_1 \leq 2], [x_2 \leq 2]$, do not hold.

This problem will be masked since because the constraint (6.1) is still in the problem as a propagator, and indeed we do not bother to save the explanation clause (6.2), we only save clauses which are *learnt from* the clause (6.2) through conflict analysis.

Through conflict analysis, the clause (6.2) can create a large family of learnt clauses which propagate when $x_1, \ldots, x_4 \leq 2$. Such clauses do not propagate and are useless in the new subproblem. A better precondition for propagation is $x_1 + \ldots + x_4 \leq 8$. □

The Multi-valued Decision Diagram (MDD) decomposition of *linear* to SAT [1] addresses this deficiency to some extent. Disregarding coefficients for simplicity, given some *linear* constraint $x_1 + \ldots + x_n \geq d$, it creates partial sum literals $[x_1 \leq u]$, $[x_1 + x_2 \leq u]$, $[x_1 + x_2 + x_3 \leq u]$, etc, for all interesting values of u.

We can also define 'virtual' partial sum literals $[x_m + \ldots + x_n \leq v], m \in 1..n$, because given the constraint $x_1 + \ldots + x_n \geq d$ holds, the literal $[x_m + \ldots + x_n \leq v]$ implies

$[x_1 + \ldots + x_{m-1} > u]$ where $u = d - v - 1$, which is the negation of an original partial sum literal defined by the MDD decomposition, $[x_1 + \ldots + x_{m-1} \leq u]$.

The MDD decomposition uses these additional partial sum literals to propagate $x_k, k \in 1..n$ with ternary explanations of the form

$$[x_1 + \ldots + x_{k-1} \leq u] \wedge [x_{k+1} + \ldots + x_n \leq v] \rightarrow [x_k \geq d - u - v].$$

This is clearly much more reuseable, since considering the original constraint as $X + x_k + Y \geq d$ where X and Y are the appropriate partial sums, we only have to fix $\mathrm{ub}(X), \mathrm{ub}(Y)$ in the clausal explanation rather than $\mathrm{ub}(x_1), \ldots, \mathrm{ub}(x_{k-1}), \mathrm{ub}(x_{k+1}), \ldots, \mathrm{ub}(x_n)$.

On the other hand, in a similar subproblem where the slack in the *linear* constraint is apprortioned differently between X and Y, the MDD explanation will not be reuseable, since it will see either $x_1 + \ldots + x_{k-1} > u$ or $x_{k+1} + \ldots + x_n > v$, whereas the correct condition for propagation would be $x_1 + \ldots + x_{k-1} + x_{k+1} + \ldots + x_n \leq u + v$.

Furthermore, the MDD decomposition requires huge numbers of clauses and partial sum literals, especially for long *linear* constraints, which is exactly the case where the *linear* propagator produces the weakest explanations.

Chu and Stuckey discuss this issue in the context of a more general discussion about improving the 'language of learning' which is the information that can be encoded into nogoods [24]. They propose an extended literal containing a *linear* equation, which would address this issue. We discuss this idea further in the next section.

The *circulation* and *min_cost_flow* propagators in Chapter 4, and the `LSSU` (branch-and-bound) optimization algorithm in Chapter 5, which are heavily based on *linear*, were quite useful in practice, but we believe they would be *very much* more effective if the issue of *linear* explanations outlined here, could be addressed.

Consider *min_cost_flow* or `LSSU` or with an objective $x_1 + \ldots + x_n$ to minimize. They are both effectively propagating the constraint $x_1 + \ldots + x_n < d$ where d is the objective of the best solution found so far. The clausal explanation is based on fixing $\mathrm{ub}(x_1), \ldots, \mathrm{ub}(x_n)$, which is not reuseable, and is weakest in the common case of very large n.

A similar issue arises with the `LSUS` algorithm, which enters assumptions $x_1 + \ldots + x_n \leq d$, as half-reified *linear* constraints, where d gradually increases until the problem is satisfiable. The `LSUS`/`LSSU`-derived `MSU3`/`4` algorithms also have exactly the same issues. Unsurprisingly, these algorithms performed badly in our experiments.

With the present *linear* explanations in LCG, the best way to achieve decent results is to bypass the need to enter any long *linear* constraints, which is exactly what the more effective `MSU1`/`WPM1`/`MaxRes`/`OLL` algorithms achieve. These algorithms, most obviously `OLL`, also use cardinality (*linear*) constraints, but they are much more targeted and hence shorter, since they only refer to the objective terms found in a given unsatisfiable core.

6.5.2 Improvements to LCG – semantic conflict analysis

Feydy et al. [45] define a semantic conflict analysis procedure in LCG based on *atoms*, essentially the literals $[x \leq v]$ and $[x = v]$, rather than uninterpreted literals. They manipulate a conflict in terms of integer variable domains directly, improving learnt clauses.

We believe that this does not go far enough. A related issue to that described in the previous section is how *linear* explanations resolve together. If this could be done semantically, then the explanations could potentially be much more reuseable.

At present, if a *linear* constraint produces a clause $C \vee [x \leq v]$ which sets $x \leq v$ because C is *false*, and another *linear* constraint produces a clause $D \vee [x > v]$ which tries to set $x > v$ because D is *false*, conflict analysis resolves $[x \leq v]$ with $[x > v]$ to create a new clause $C \vee D$ which is also *false*. This cuts off the current subproblem without reference to x, which saves us the work of propagating x in future.

Suppose some *linear* constraint $X + x \leq d$ propagates to set $x \leq d - \text{ub}(X)$, and another *linear* constraint $Y + x \geq e$ tries to propagate to set $x \geq e - \text{lb}(Y)$ which is in conflict, where X and Y are arbitrary linear combinations of variables. With the approach we propose here, semantic conflict analysis could subtract the two constraints to produce a *linear* resolvent $X - Y \leq d - e$ which is *false* if $e - \text{lb}(Y) > d - \text{ub}(X)$.

We constructed several experimental Pseudo-Boolean solvers using this approach with various modifications. We hoped to extend Santos and Manquinho's [97] and Dixon and Ginsberg [38]'s work to integer variables and hence LCG, but this line of research has not returned useful results to date. We had to put this aside due to time constraints.

One problem with the above occurs when the *linear* equation has coefficients, ignored in the previous examples for simplicity. The *linear* propagator sets $x \leq v$ or $x \geq v$ for fractional values of v, which it aggressively rounds to the appropriate integer. Unfortunately two *linear* constraints which are separately *false* after rounding can produce a resolvent which is not *false* after rounding, requiring significant workarounds [38, 97].

Nieuwenhuis has a comprehensive discussion of this problem [81], and a complicated approach which is claimed to work well. We propose the following much simpler semantic conflict analysis scheme. We can substitute $x > v$ into the first equation and $x \leq v$ into the second equation, producing equivalent but slightly weaker half-reified constraints,

$$[x > v] \rightarrow (X < d - v) \quad \text{and} \quad [x \leq v] \rightarrow (Y \geq e - v).$$

These resolve together by considering that either $[x > v]$ or $[x \leq v]$ holds, giving the semantic resolvent $(X < d - v) \vee (Y \geq e - v)$. This could then be re-encoded back into LCG and returned to the problem as effectively a learnt clause.

6.5.3 Improvements to LCG – semantic unsatisfiable cores

Another area we would like to investigate is to extend or redesign our unsatisfiable-core optimization algorithms from Chapter 5 to use Feydy et al.'s semantic conflict analysis [45] to enable better information to be extracted from unsatisfiable cores.

Suppose x is a variable in the objective (possibly with a coefficient that we ignore for now). We solve under the assumption $[x \leq v]$ for some $v \in D(x)$, where initially $v = \text{lb}(x)$. If this is infeasible, the unsatisfiable core potentially contains $[x \leq v]$.

Given the unsatisfiable core contains only assumptions, if a problem constraint explains itself by a literal like $[x \leq w] \wedge \ldots \rightarrow \ldots$, where $w > v$, then the final conflict analysis given in Algorithm 2.3 would explain this by $[x \leq v] \rightarrow [x \leq w]$, which unnecessarily weakens the unsatisfiable core to contain $[x \leq v]$ rather than the original $[x \leq w]$.

We should use semantic unsatisfiable cores containing original literals like $[x \leq w]$ by which the propagators explained themselves, instead of assumptions like $[x \leq v]$. We should use Feydy et al.'s procedure to make w as large as possible. Then our optimization algorithms could make more progress on each iteration, giving faster convergence.

To illustrate the idea, consider plain `LSUS` without unsatisfiable cores. We start with an objective estimate $z_{lb} = \text{lb}(z)$ and solve under the assumption $[z \leq z_{lb}]$, increasing z_{lb} by 1 each time this fails. If semantic conflict analysis returned a meaningful literal like $[z > w]$ then we could set z_{lb} to $w + 1$, potentially a large jump saving many iterations.

We now consider applying this approach to `WPM1`. Let A be an assumptions map where initially $A(x) = \text{lb}(x)$ for each x in the objective. At each iteration we solve under assumptions $[x \leq A(x)]$ obtaining an unsatisfiable core $[x_1 \leq v_1] \wedge \ldots \wedge [x_n \leq v_n] \rightarrow \mathit{false}$.

To utilize a semantic unsatisfiable core with `WPM1`, we propose to calculate $w_{min} = \min_{i=1}^{n}(v_n + 1 - A(n))$, and take this as the increase to the objective z_{lb} and all assumptions $A(x_1), \ldots, A(x_n)$. We would then add clauses similarly to `WPM1`, stating that no more than one of the *original* assumptions $[x_i \leq A(x_i)]$ can be relaxed simultaneously.

This gives correct results on the next iteration, but a problem occurs further down the track. Initially `WPM1` tries relaxing just one assumption literal from the unsatisfiable core, which is correct. If this has no solution it tries relaxing another and so on, which is incorrect, since only the first literal can have its assumption relaxed by w_{min}. The value w_{min} might be too large for the second and subsequent relaxations of the same unsatisfiable core.

Example 6.2 Consider maximizing $x+y$ where $x, y \in 0..4$ subject to $2x+3y \geq 8$. Figure 6.1a shows the feasible region of this problem. We solve under assumptions $[x \leq 0] \wedge [y \leq 0]$. Currently this returns an unsatisfiable core $[x \leq 0] \wedge [y \leq 0] \rightarrow \mathit{false}$. One example of a semantic unsatisfiable core would be $[x \leq 2] \wedge [y \leq 1] \rightarrow \mathit{false}$ which excludes the region shown in Figure 6.1b. Observe that this does not intersect the feasible region.

Then $[x > 2]$ or $[y > 1]$ holds. So x increases by 3 from its current assumption $A(x) = 0$, or y increases by 2 from its current assumption $A(y) = 0$. We set $w_{min} = 2$ being the minimum

y \ x	0	1	2	3	4
0	0	2	4	6	8
1	3	5	7	9	11
2	6	8	10	12	14
3	9	11	13	15	17
4	12	14	16	18	20

(a) Showing $2x + 3y \geq 8$

y \ x	0	1	2	3	4
0	0	2	4	6	8
1	3	5	7	9	11
2	6	8	10	12	14
3	9	11	13	15	17
4	12	14	16	18	20

(b) 1st core $[x \leq 2] \wedge [y \leq 1]$

y \ x	0	1	2	3	4
0	0	2	4	6	8
1	3	5	7	9	11
2	6	8	10	12	14
3	9	11	13	15	17
4	12	14	16	18	20

(c) Next region considered

Fig. 6.1 The effect of a semantic unsatisfiable core on the solution set considered

of these increases, giving a normalized unsatisfiable core $[x \leq 1] \wedge [y \leq 1] \rightarrow \mathit{false}$, relaxed assumptions $A(x) = A(y) = 2$, and an objective increase $z_{lb} = 2$.

We create a new integer variable $j \in 1..2$ and Booleans x', y', then post new constraints

$$[x > 0] \wedge [j \neq 1] \rightarrow x' \quad \text{and} \quad [y > 0] \wedge [j \neq 2] \rightarrow y',$$

and add fresh assumptions $A(x') = A(y') = 0$. The revised assumptions set $[x \leq 2] \wedge [y \leq 2] \wedge \neg x' \wedge \neg y'$ opens up the region shown in Figure 6.1c for consideration, which is correct noting that none of these cells have objective greater than $z_{lb} = 2$.

A subsequent unsatisfiable core like $[x \leq 3] \wedge \neg y' \rightarrow \mathit{false}$ requires us to relax y', but relaxing y' allows *both* of the literals $[x \leq 1]$ and $[y \leq 1]$ in the original semantic unsatisfiable core to be *false*, leading to both x and y increasing by the original $w_{min} = 2$, and opening up the suboptimal solution $x = 2, y = 2$ with objective $z = 4$. □

One possible solution is, instead of creating Boolean variables to catch the case that multiple assumptions in the unsatisfiable core must be relaxed simultaneously (as `WPM1` and `MaxRes` would do), we can create an integer variable as a violation measure for each assumption. These would be initially assumed zero and relaxed only gradually.

Example 6.3 Referring to the previous example, instead of Boolean variables x', y', consider creating integer variables $x', y' \in 0..2$, then posting the following constraints,

$$[x > 0] \wedge [j \neq 1] \rightarrow [x' > 0], \qquad [y > 0] \wedge [j \neq 2] \rightarrow [y' > 0],$$
$$[x > 1] \wedge [j \neq 1] \rightarrow [x' > 1], \qquad [y > 1] \wedge [j \neq 2] \rightarrow [y' > 1],$$

and solving under assumptions $[x \leq 2], [y \leq 2], [x' \leq 0], [y' \leq 0]$. Then an unsatisfiable core like $[x \leq 3] \wedge [y' \leq 0] \rightarrow \mathit{false}$ would only increase the y' assumption to $[y' \leq 1]$ giving a solution space like $(x, y) \in 0..2 \times 0..1$ without suboptimal solutions. □

We might also consider extending `OLL` to semantic unsatisfiable cores, in which the new integer variable added by `OLL` is extended to be a violation measure considering all variables in the unsatisfiable core, rather than just a violation count over the literals.

Alternatively, we can define a native LCG version of `WPM1` or `MaxRes` in which the violation-measure variables actually replace the old variables in the objective. We can simply

calculate a set of new objective terms from those in the unsatisfiable core, identical except that one of the new terms is w_{min} less than its corresponding original term.

Example 6.4 Given an objective $x + y$ where $x, y \in 0..4$ and a normalized unsatisfiable core $[x \leq 2] \wedge [y \leq 2] \rightarrow \mathit{false}$, create new variables $j \in 1..2, x', y' \in 0..4$ and post

$$x' \geq x - 2[j = 1] \quad \text{and} \quad y' \geq y - 2[j = 2].$$

We can then proceed to minimize the new objective $x' + y'$ instead of $x + y$. □

We implemented this in *CPX* which has semantic conflict analysis. It was not competitive with *MSUnCore*'s `MSU1` implementation at the time, but since then we have learned how to do much more efficient implementations of unsatisfiable-core optimization, so we should try this again. In any case, we clearly should continue trying to develop native LCG unsatisfiable-core algorithms, rather than just MaxSAT algorithms adapted to LCG.

6.5.4 Improvements to LCG – LNS and inference based neighbourhoods

In our early attempts to solve industrial problems with LCG, we would encounter a problem like Car Sequencing [105], Nurse Scheduling Problems [113], etc, download a suite of huge standard instances, and find LCG making little progress. Since we generate proveably optimal solutions as opposed to the standard 'merely good' solutions, we had to try smaller instances, which was unfortunate, even though our solutions are stronger.

A more robust approach to this issue would be to implement localsearch techniques in LCG. The most promising of these is Large Neighbourhood Search (LNS) [86, 103] or its relative Very Large Scale Neighbourhoods (VLSN) [4], which seem ideal for LCG. This would allow tackling large industrial scheduling instances on an equal footing with established algorithms which produce 'merely good' solutions.

We constructed LNS algorithms using our previous prototype solver CPX [45] for Car Sequencing, Nurse Scheduling Problems, and several other problems tackled in this thesis. We used a solution pool to avoid being trapped near local minima. We generated the neighbourhoods by deleting n arbitrary nurses' schedules from a pool solution, or by deleting up to two disjoint blocks of n consecutive days (for all nurses) or cars.

Our LNS approach did not converge to good solutions as quickly as we had hoped, so we would like to put more research into good neighbourhood selection. In particular, since LCG discovers inferences between variables, it should be able to define good neighbourhoods based on learnt clauses. This could be *much* better than LNS in traditional CP.

More recently Rendl et al. announced MiniSearch [93] as an extension of MiniZinc [80], which promises to add LNS to LCG in an elegant way. We would certainly like to try our models with the randomized or adaptive LNS offered by MiniSearch, and then try to improve the neighbourhood selection further based on learnt clauses.

6.5.5 Applications of LCG – Minimum Spanning Trees

We would like to do more work on Network Flows vs. Spanning Trees with learning, following de Uña et al. [35]. We tried the Hop-Constrained Minimum Spanning Tree problem (HMST) [56] problem using *min_cost_flow* to enforce the hop-count and a Pseudo-Boolean objective to enforce minimality, and this showed a minor improvement over a naive model, but it was not significant enough to bother reporting on.

We would like to try extending our Network Flow work to solve Minimum Spanning Trees (MST) efficiently. We propose an application for this in Printed Circuit Board (PCB) design, using network flows for so-called *channel routing*, i.e. the rough routing based on the width of a channel between IC pins and therefore how many tracks it can support, plus MST constraints for fine grained routing under a length minimization objective.

6.5.6 Applications of LCG – 'C' program verification

Another idea we tackled, which has not yet resulted in a publication, is using LCG for 'C' program verification, as a backend to the CBMC bounded model checker [27].

We produced a working prototype, using unsatisfiable cores to generate error locations which are subsets of program lines responsible for a violated assertion [59], but we discovered that 'C' programs are not well behaved enough to model using CP constraints, so we would like to have a theory of bitvectors, similar to that offered by SMT solvers.

There is recent progress in CP/SAT with bitvectors [75, 115]. We would like to try creating an LCG-like solver which uses a Boolean model of bitvector (instead of integer) domains, with propagators for the common operations used by 'C' programs.

An interesting LCG propagator in our error localization prototype is a Difference Logic propagator with explanations, based on Feydy et al.'s [43] which does not have explanations. The Difference Logic LP is the dual of Minimum Cost Flow, so we created an experimental propagator based on *min_cost_flow*, implementing a subset of Feydy et al.'s propagator. It should be migrated into our latest solver, completed, and evaluated.

6.6 Final remarks

We set out to encode some large, realistic scheduling and rostering problems into CP and try to solve them using LCG techniques, and to extend the basic LCG solver in whatever direction necessary to solve these kinds of problems efficiently.

We certainly accomplished this goal with the unsatisfiable-core work, which leverages the unique capability of LCG to analyze the interaction of constraints and derive proofs of unsatisfiability. We accomplished the goal to some extent by adding new explanation-capable propagators to LCG, although the approach of simply adding more and more expensive

propagation algorithms to LCG delivers diminishing returns, due to the unique capability of LCG to use cheaper propagators or decompositions efficiently.

CP is quite a young discipline and LCG is clearly a very fertile area within CP. While this research has certainly returned some very promising initial results, we feel there is still much to do. In the course of this research, we became very familiar with the strengths and limitations of LCG as applied to realistic scheduling and rostering problems, which in turn suggested many interesting future modifications to the basic solver.

Bibliography

1. Abío, I., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E., Stuckey, P.J.: To encode or to propagate? the best choice for each constraint in SAT. In: C. Schulte (ed.) *Principles and Practice of Constraint Programming: 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, pp. 97–106. Springer Berlin Heidelberg (2013)
2. Achterberg, T.: Conflict analysis in mixed integer programming. *Discrete Optimization 4*(1), 4–20 (2007). IMA Special Workshop on Mixed-Integer Programming
3. Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network Flows: Theory, Algorithms, and Applications. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1993)
4. Ahuja, R.K., Ergun, Ö., Orlin, J.B., Punnen, A.P.: A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics 123*(1–3), 75–102 (2002)
5. Andres, B., Kaufmann, B., Mattheis, O., Schaub, T.: Unsatisfiability-based optimization in clasp. In: A. Dovier, V.S. Costa (eds.) *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, September 4-8, 2012, Budapest, Hungary*, *LIPIcs*, vol. 17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2012)
6. Ansótegui, C., Bonet, M.L., Levy, J.: Solving (weighted) partial MaxSAT through satisfiability testing. In: O. Kullmann (ed.) *Theory and Applications of Satisfiability Testing - SAT 2009*, *Lecture Notes in Computer Science*, vol. 5584, pp. 427–440. Springer Berlin Heidelberg (2009)
7. Ansótegui, C., Bonet, M.L., Gabàs, J., Levy, J.: Improving SAT-based weighted MaxSAT solvers. In: M. Milano (ed.) *Principles and Practice of Constraint Programming: 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, pp. 86–101. Springer Berlin Heidelberg (2012)
8. Argelich, J., Li, C.M., Manyà, F., Planes, J., Martins, R.: Ninth Max-SAT evaluation (2014). URL `http://maxsat.ia.udl.cat`
9. Asín Achá, R., Nieuwenhuis, R.: Curriculum-based course timetabling with SAT and MaxSAT. *Annals of Operations Research 218*(1), 71–91 (2014)
10. Audemard, G., Lagniez, J.M., Simon, L.: Improving glucose for incremental SAT solving with assumptions: Application to mus extraction. In: M. Järvisalo, A. van Gelder (eds.)

Theory and Applications of Satisfiability Testing – SAT 2013: 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings, pp. 309–317. Springer Berlin Heidelberg (2013)

11. Becket, R., Brand, S., Brown, M., Duck, G.J., Feydy, T., Fischer, J., Huang, J., Marriott, K., Nethercote, N., Puchinger, J., Rafeh, R., Stuckey, P.J., Wallace, M.G.: The many roads leading to Rome: Solving Zinc models by various solvers. In: *7th International Workshop on Constraint Modelling and Reformulation (ModRef-08)* (2008). URL `https://www.it.uu.se/research/group/astra/ModRef08/G12.pdf`
12. Beldiceanu, N., Carlsson, M., Demassey, S., Petit, T.: Global constraint catalogue: Past, present and future. *Constraints 12*(1), 21–62 (2007)
13. Belov, A., Marques-Silva, J.: MUSer2: An efficient MUS extractor. *JSAT 8*(3/4), 123–128 (2012)
14. Belov, G., Stuckey, P.J., Tack, G., Wallace, M.: Improved linearization of constraint programming models. In: M. Rueher (ed.) *Principles and Practice of Constraint Programming: 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, pp. 49–65. Springer International Publishing, Cham (2016)
15. Berge, C., Ghouila-Houri, A.: Programming, games and transportation networks. London, Methuen (1965)
16. Biere, A.: PicoSAT essentials. *JSAT 4*(2-4), 75–97 (2008)
17. Bjork, M.: Successful SAT encoding techniques. *JSAT Addendum 2009-07-25* (2009). URL `http://jsat.ewi.tudelft.nl/addendum/Bjork_encoding.pdf`
18. Bjorner, N., Phan, A.D.: *νZ* - maximal satisfaction with Z3. In: T. Kutsia, A. Voronkov (eds.) *SCSS 2014. 6th International Symposium on Symbolic Computation in Software Science*, *EPiC Series in Computing*, vol. 30, pp. 1–9. EasyChair (2014)
19. Bockmayr, A., Pisaruk, N., Aggoun, A.: Network flow problems in constraint programming. In: T. Walsh (ed.) *Principles and Practice of Constraint Programming – CP 2001*, *Lecture Notes in Computer Science*, vol. 2239, pp. 196–210. Springer Berlin Heidelberg (2001)
20. Bonet, M.L., Levy, J., Manyà, F.: Resolution for Max-SAT. *Artificial Intelligence 171*(8-9), 606–618 (2007)
21. Choi, C.W., Harvey, W., Lee, J.H.M., Stuckey, P.J.: Finite domain bounds consistency revisited. In: A. Sattar, B.h. Kang (eds.) *AI 2006: Advances in Artificial Intelligence: 19th Australian Joint Conference on Artificial Intelligence, Hobart, Australia, December 4-8, 2006. Proceedings*, pp. 49–58. Springer Berlin Heidelberg (2006)
22. Chu, G., Stuckey, P.J.: Chuffed - a lazy clause solver. Tech. rep., National ICT Australia and The University of Melbourne, Australia (2010). URL `http://www.minizinc.org/challenge2010/description_chuffed.txt`. See also `https://github.com/geoffchu/chuffed`

23. Chu, G., Stuckey, P.J.: Inter-instance nogood learning in constraint programming. In: M. Milano (ed.) *Principles and Practice of Constraint Programming: 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, pp. 238–247. Springer Berlin Heidelberg (2012)
24. Chu, G., Stuckey, P.J.: Structure based extended resolution for constraint programming. *CoRR abs/1306.4418* (2013). URL `http://arxiv.org/abs/1306.4418`
25. Chu, G., Garcia de la Banda, M., Mears, C., Stuckey, P.J.: Symmetries, almost symmetries, and lazy clause generation. *Constraints 19*(4), 434–462 (2014)
26. Chvátal, V.: Linear Programming. Freeman (1983)
27. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: K. Jensen, A. Podelski (eds.) *Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004. Proceedings*, pp. 168–176. Springer Berlin Heidelberg (2004)
28. Colton, S., Miguel, I.: Constraint generation via automated theory formation. In: T. Walsh (ed.) *Principles and Practice of Constraint Programming — CP 2001: 7th International Conference, CP 2001 Paphos, Cyprus, November 26 – December 1, 2001 Proceedings*, pp. 575–579. Springer Berlin Heidelberg (2001)
29. Costa, M.C.: Persistency in maximum cardinality bipartite matchings. *Operations Research Letters 15*(3), 143–149 (1994)
30. Data61 CSIRO: MiniZinc challenge (2016). URL `http://www.minizinc.org/challenge2016/challenge.html`
31. Davey, B., Boland, N., Stuckey, P.J.: Efficient intelligent backtracking using linear programming. *INFORMS Journal on Computing 14*(4), 373–386 (2002)
32. Davis, M., Putnam, H.: A computing procedure for quantification theory. *J. ACM 7*(3), 201–215 (1960)
33. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. *Commun. ACM 5*(7), 394–397 (1962)
34. de Moura, L., Bjørner, N.: Satisfiability modulo theories: An appetizer. In: M.V.M. Oliveira, J. Woodcock (eds.) *Formal Methods: Foundations and Applications: 12th Brazilian Symposium on Formal Methods, SBMF 2009 Gramado, Brazil, August 19-21, 2009 Revised Selected Papers*, pp. 23–36. Springer Berlin Heidelberg (2009)
35. de Uña, D., Gange, G., Schachte, P., Stuckey, P.J.: Steiner tree problems with side constraints using constraint programming. In: *Proceedings of the Thirtieth Conference on Artificial Intelligence*, *AAAI '16*, pp. 3383–3389. Association for the Advancement of Artificial Intelligence, Palo Alto, CA, USA (2016)
36. Dechter, R.: Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence 41*(3), 273–312 (1990)

37. Di Gaspero, L., McCollum, B., Schaerf, A.: The second international timetabling competition (ITC-2007): Curriculum-based course timetabling (track 3). Tech. rep., University of Udine, Italy; Queens University, Belfast, United Kingdom (2007). URL http://www.cs.qub.ac.uk/itc2007/curriculmcourse/report/curriculumtechreport.pdf
38. Dixon, H., Ginsberg, M.L.: Inference methods for a pseudo-Boolean satisfiability solver. In: *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, *AAAI'02*, pp. 635–640. AAAI Press (2002). URL https://www.aaai.org/Papers/AAAI/2002/AAAI02-095.pdf
39. Edmonds, J., Karp, R.M.: Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM 19*(2), 248–264 (1972)
40. Eén, N., Sörensson, N.: An extensible SAT-solver. In: E. Giunchiglia, A. Tacchella (eds.) *Theory and Applications of Satisfiability Testing*, *Lecture Notes in Computer Science*, vol. 2919, pp. 502–518. Springer Berlin Heidelberg (2004)
41. Feydy, T., Stuckey, P.J.: Lazy clause generation reengineered. In: I.P. Gent (ed.) *Principles and Practice of Constraint Programming - CP 2009*, *Lecture Notes in Computer Science*, vol. 5732, pp. 352–366. Springer Berlin Heidelberg (2009)
42. Feydy, T., Stuckey, P.J.: Interval constraints with learning: Application to air traffic control. In: M. Rueher (ed.) *Principles and Practice of Constraint Programming: 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, pp. 224–232. Springer International Publishing, Cham (2016)
43. Feydy, T., Schutt, A., Stuckey, P.J.: Global difference constraint propagation for finite domain solvers. In: *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, *PPDP '08*, pp. 226–235. ACM, New York, NY, USA (2008)
44. Feydy, T., Somogyi, Z., Stuckey, P.J.: Half reification and flattening. In: J. Lee (ed.) *Principles and Practice of Constraint Programming – CP 2011: 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, pp. 286–301. Springer Berlin Heidelberg (2011)
45. Feydy, T., Schutt, A., Stuckey, P.: Semantic learning for lazy clause generation. In: *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2013* (2013). URL http://cp2013.a4cp.org/sites/default/files/uploads/trics2013_submission_7.pdf
46. Ford, L., Fulkerson, D.: Flows in Networks. *Rand Corporation research study*. University Press (1962)
47. Francis, K.G., Stuckey, P.J.: Explaining circuit propagation. *Constraints 19*(1), 1–29 (2014)
48. Fu, Z., Malik, S.: On solving the partial MAX-SAT problem. In: A. Biere, C.P. Gomes (eds.) *Theory and Applications of Satisfiability Testing - SAT 2006: 9th International Conference, Seattle, WA, USA, August 12-15, 2006. Proceedings*, pp. 252–265. Springer

Berlin Heidelberg (2006)
49. Gange, G., Stuckey, P.J., Szymanek, R.: MDD propagators with explanation. *Constraints 16*(4), 407–429 (2011)
50. Gange, G., Stuckey, P.J., van Hentenryck, P.: Explaining propagators for edge-valued decision diagrams. In: C. Schulte (ed.) *Principles and Practice of Constraint Programming*, *Lecture Notes in Computer Science*, vol. 8124, pp. 340–355. Springer Berlin Heidelberg (2013)
51. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: clasp: A conflict-driven answer set solver. In: C. Baral, G. Brewka, J. Schlipf (eds.) *Logic Programming and Nonmonotonic Reasoning: 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007. Proceedings*, pp. 260–265. Springer Berlin Heidelberg (2007)
52. Gecode Team: Gecode: Generic constraint development environment (2015). `http://www.gecode.org`
53. Gent, I.P., Walsh, T.: CSPlib: A benchmark library for constraints. In: J. Jaffar (ed.) *Principles and Practice of Constraint Programming – CP99*, *Lecture Notes in Computer Science*, vol. 1713, pp. 480–481. Springer Berlin Heidelberg (1999). URL `http://www.csplib.org`
54. Gent, I.P., Miguel, I., Nightingale, P.: Generalised arc consistency for the alldifferent constraint: An empirical survey. *Artificial Intelligence 172*(18), 1973–2000 (2008)
55. Gomes, C.P., Shmoys, D.: Completing quasigroups or latin squares: A structured graph coloring problem. In: *Proc. Computational Symposium on Graph Coloring and Generalizations* (2002)
56. Gouveia, L.: Multicommodity flow models for spanning trees with hop constraints. *European Journal of Operational Research 95*(1), 178–190 (1996)
57. Hall, P.: On representatives of subsets. *Journal of the London Mathematical Society s1-10*(1), 26–30 (1935)
58. Hopcroft, J.E., Karp, R.M.: An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing 2*(4), 225–231 (1973)
59. Jose, M., Majumdar, R.: Cause clue clauses: Error localization using maximum satisfiability. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, *PLDI '11*, pp. 437–446. ACM, New York, NY, USA (2011)
60. Jussien, N., Barichard, V.: The PaLM system: explanation-based constraint programming. In: *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, pp. 118–133 (2000)
61. Katsirelos, G.: Nogood processing in CSPs. Ph.D. thesis, University of Toronto, Canada (2008)
62. Katsirelos, G., Bacchus, F.: Unrestricted nogood recording in CSP search. In: F. Rossi (ed.) *Principles and Practice of Constraint Programming – CP 2003: 9th International Conference, CP 2003, Kinsale, Ireland, September 29 – October 3, 2003. Proceedings*, pp.

873–877. Springer Berlin Heidelberg (2003)
63. Kolisch, R., Sprecher, A.: PSPLIB - a project scheduling problem library: OR software - ORSEP operations research software exchange program. *European Journal of Operational Research 96*(1), 205 – 216 (1997)
64. Lopes, L., Smith-Miles, K.: Pitfalls in instance generation for Udine timetabling. In: C. Blum, R. Battiti (eds.) *Learning and Intelligent Optimization: 4th International Conference, LION 4, Venice, Italy, January 18-22, 2010. Selected Papers*, pp. 299–302. Springer Berlin Heidelberg (2010)
65. Lopez-Ortiz, A., Quimper, C.G., Tromp, J., van Beek, P.: A fast and simple algorithm for bounds consistency of the all different constraint. In: *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, *IJCAI'03*, pp. 245–250. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2003)
66. Lovász, L., Plummer, M.D.: Matching Theory. North Holland (1986)
67. Maher, M., Narodytska, N., Quimper, C.G., Walsh, T.: Flow-based propagators for the SEQUENCE and related global constraints. In: P.J. Stuckey (ed.) *Principles and Practice of Constraint Programming: 14th International Conference, CP 2008, Sydney, Australia, September 14-18, 2008. Proceedings*, pp. 159–174. Springer Berlin Heidelberg (2008)
68. Malik, A.M., Chase, M., Russell, T., van Beek, P.: An application of constraint programming to superblock instruction scheduling. In: P.J. Stuckey (ed.) *Principles and Practice of Constraint Programming: 14th International Conference, CP 2008, Sydney, Australia, September 14-18, 2008. Proceedings*, pp. 97–111. Springer Berlin Heidelberg (2008)
69. Manquinho, V., Marques-Silva, J., Planes, J.: Algorithms for weighted Boolean optimization. In: O. Kullmann (ed.) *Theory and Applications of Satisfiability Testing - SAT 2009: 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, pp. 495–508. Springer Berlin Heidelberg (2009)
70. Marques-Silva, J.: The MSUnCore MaxSAT solver. Tech. rep., University College Dublin, Ireland (2009). URL `http://logos.ucd.ie/~jpms/pubs/pubsite/msu-booklet09.pdf`
71. Marques-Silva, J., Planes, J.: On using unsatisfiability for solving maximum satisfiability. *CoRR abs/0712.1097* (2007). URL `http://arxiv.org/abs/0712.1097`
72. Marques-Silva, J., Planes, J.: Algorithms for maximum satisfiability using unsatisfiable cores. In: *Proceedings of the Conference on Design, Automation and Test in Europe*, *DATE '08*, pp. 408–413. ACM, New York, NY, USA (2008)
73. Marques Silva, J., Sakallah, K.: GRASP–a new search algorithm for satisfiability. In: *Computer-Aided Design, 1996. ICCAD-96. Digest of Technical Papers., 1996 IEEE/ACM International Conference on*, pp. 220–227 (1996)
74. Marques-Silva, J., Sakallah, K.: GRASP: A search algorithm for propositional satisfiability. *Computers, IEEE Transactions on 48*(5), 506–521 (1999)

75. Michel, L.D., van Hentenryck, P.: Constraint satisfaction over bit-vectors. In: M. Milano (ed.) *Principles and Practice of Constraint Programming: 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, pp. 527–543. Springer Berlin Heidelberg (2012)
76. Morgado, A., Heras, F., Liffiton, M., Planes, J., Marques-Silva, J.: Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints 18*(4), 478–534 (2013)
77. Morgado, A., Dodaro, C., Marques-Silva, J.: Core-guided MaxSAT with soft cardinality constraints. In: B. O'Sullivan (ed.) *Principles and Practice of Constraint Programming*, *Lecture Notes in Computer Science*, vol. 8656, pp. 564–573. Springer International Publishing (2014)
78. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: *Proceedings of the 38th Annual Design Automation Conference*, *DAC '01*, pp. 530–535. ACM, New York, NY, USA (2001)
79. Narodytska, N., Bacchus, F.: Maximum satisfiability using core-guided MaxSAT resolution. In: *Proceedings of the Twenty-Eighth Conference on Artificial Intelligence*, *AAAI '14*, pp. 2717–2723. Association for the Advancement of Artificial Intelligence, Palo Alto, CA, USA (2014)
80. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: C. Bessière (ed.) *Principles and Practice of Constraint Programming – CP 2007*, *Lecture Notes in Computer Science*, vol. 4741, pp. 529–543. Springer Berlin Heidelberg (2007)
81. Nieuwenhuis, R.: The IntSat method for integer linear programming. In: B. O'Sullivan (ed.) *Principles and Practice of Constraint Programming: 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, pp. 574–589. Springer International Publishing, Cham (2014)
82. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM 53*(6), 937–977 (2006)
83. Ohrimenko, O., Stuckey, P.J., Codish, M.: Propagation via lazy clause generation. *Constraints 14*(3), 357–391 (2009)
84. Ortega, F., Wolsey, L.A.: A branch-and-cut algorithm for the single-commodity, uncapacitated, fixed-charge network flow problem. *Networks 41*(3), 143–158 (2003)
85. Pipatsrisawat, K., Darwiche, A.: Rsat 1.03: SAT solver description. Tech. rep., University of California, Los Angeles (2007). URL `http://reasoning.cs.ucla.edu/rsat/papers/rsat_1.03.pdf`
86. Pisinger, D., Ropke, S.: Large neighborhood search. In: M. Gendreau, J.Y. Potvin (eds.) *Handbook of Metaheuristics*, pp. 399–419. Springer US, Boston, MA (2010)
87. Prud'homme, C., Fages, J.G., Lorca, X.: Choco Documentation. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S. (2015). URL `http://www.choco-solver.`

org

88. Puget, J.F.: A fast algorithm for the bound consistency of alldiff constraints. In: *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, *AAAI'98*, pp. 359–366. AAAI Press (1998)
89. Quimper, C.G., van Beek, P., López-Ortiz, A., Golynski, A., Sadjad, S.B.: An efficient bounds consistency algorithm for the global cardinality constraint. In: F. Rossi (ed.) *Principles and Practice of Constraint Programming – CP 2003: 9th International Conference, CP 2003, Kinsale, Ireland, September 29 – October 3, 2003. Proceedings*, pp. 600–614. Springer Berlin Heidelberg (2003)
90. Régin, J.C.: A filtering algorithm for constraints of difference in CSPs. In: *Proceedings of the Twelfth National Conference on Artificial Intelligence (Vol. 1)*, *AAAI '94*, pp. 362–367. American Association for Artificial Intelligence, Menlo Park, CA, USA (1994)
91. Régin, J.C.: Generalized arc consistency for global cardinality constraint. In: *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 1*, *AAAI'96*, pp. 209–215. AAAI Press (1996)
92. Régin, J.C., Puget, J.F.: A filtering algorithm for global sequencing constraints. In: G. Smolka (ed.) *Principles and Practice of Constraint Programming-CP97: Third International Conference, CP97 Linz, Austria, October 29 – November 1, 1997 Proceedings*, pp. 32–46. Springer Berlin Heidelberg (1997)
93. Rendl, A., Guns, T., Stuckey, P.J., Tack, G.: MiniSearch: A solver-independent meta-search language for MiniZinc. In: G. Pesant (ed.) *Principles and Practice of Constraint Programming: 21st International Conference, CP 2015, Cork, Ireland, August 31 – September 4, 2015, Proceedings*, pp. 376–392. Springer International Publishing, Cham (2015)
94. Rochart, G.: Explications et programmation par contraintes avancée. Ph.D. thesis, Université de Nantes, France (2005)
95. Rochart, G., Jussien, N., Laburthe, F.: Challenging explanations for global constraints. In: B. O'Sullivan (ed.) *Proceedings of the third International Workshop on User Interaction in Constraint Satisfaction (UICS'03)*, pp. 31–43. Cork University (2003). URL `http://www.cs.ucc.ie/~osullb/UICS-03/uics2003proceedings.pdf`
96. Sanchez, M., Bouveret, S., de Givry, S., Heras, F., Jégou, P., Larrosa, J., Ndiaye, S., Rollon, E., Schiex, T., Terrioux, C., Verfaillie, G., Zytnicki, M.: Max-CSP competition 2008: toulbar2 solver description. Tech. rep., INRA, Toulouse, France; UPC, Barcelona, Spain; ONERA, Toulouse, France; LSIS, Marseilles, France (2008). URL `https://www.researchgate.net/publication/242297950_Max-CSP_competition_2008_toulbar2_solver_description`
97. Santos, J., Manquinho, V.: Learning techniques for pseudo-Boolean solving. In: P. Rudnicki, G. Sutcliffe, B. Konev, R. Schmidt, S. Schulz (eds.) *7th International Workshop on the Implementation of Logics (IWIL-08), a workshop of 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-08). Proceedings*

(2008). URL http://ceur-ws.org/Vol-418/paper8.pdf

98. Schutt, A., Stuckey, P.J.: Explaining producer/consumer constraints. In: M. Rueher (ed.) *Principles and Practice of Constraint Programming: 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, pp. 438–454. Springer International Publishing, Cham (2016)
99. Schutt, A., Feydy, T., Stuckey, P.J., Wallace, M.G.: Explaining the cumulative propagator. *Constraints 16*(3), 250–282 (2011)
100. Schutt, A., Stuckey, P.J., Verden, A.R.: Optimal carpet cutting. In: J. Lee (ed.) *Principles and Practice of Constraint Programming – CP 2011: 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, pp. 69–84. Springer Berlin Heidelberg (2011)
101. Schutt, A., Feydy, T., Stuckey, P.J.: Explaining time-table-edge-finding propagation for the cumulative resource constraint. In: C. Gomes, M. Sellmann (eds.) *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, *Lecture Notes in Computer Science*, vol. 7874, pp. 234–250. Springer Berlin Heidelberg (2013)
102. Schutt, A., Feydy, T., Stuckey, P.J., Wallace, M.G.: Solving RCPSP/max by lazy clause generation. *Journal of Scheduling 16*(3), 273–289 (2013)
103. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. In: M. Maher, J.F. Puget (eds.) *Principles and Practice of Constraint Programming — CP98: 4th International Conference, CP98 Pisa, Italy, October 26–30, 1998 Proceedings*, pp. 417–431. Springer Berlin Heidelberg (1998)
104. Simonis, H.: Kakuro as a constraint problem. In: P. Flener, H. Simonis (eds.) *7th International Workshop on Constraint Modelling and Reformulation (ModRef-08)*. Uppsala University, Computing Science (2008)
105. Solnon, C., Cung, V.D., Nguyen, A., Artigues, C.: The car sequencing problem: Overview of state-of-the-art methods and industrial case-study of the ROADEF'2005 challenge problem. *European Journal of Operational Research 191*(3), 912–927 (2008)
106. Steiger, R., van Hoeve, W.J., Szymanek, R.: An efficient generic network flow constraint. In: *Proceedings of the 2011 ACM Symposium on Applied Computing*, *SAC '11*, pp. 893–900. ACM, New York, NY, USA (2011)
107. Tamura, N., Taga, A., Kitagawa, S., Banbara, M.: Compiling finite linear CSP into SAT. In: F. Benhamou (ed.) *Principles and Practice of Constraint Programming – CP 2006*, *Lecture Notes in Computer Science*, vol. 4204, pp. 590–603. Springer Berlin Heidelberg (2006)
108. Tarjan, R.: Depth-first search and linear graph algorithms. *SIAM Journal on Computing 1*(2), 146–160 (1972)
109. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. *J. ACM 22*(2), 215–225 (1975)

110. van Hentenryck, P., Saraswat, V., Deville, Y.: Design, implementation, and evaluation of the constraint language cc(FD). *The Journal of Logic Programming 37*(1–3), 139–164 (1998)
111. van Hoeve, W.J.: The alldifferent constraint: A survey. *CoRR cs.PL/0105015* (2001). URL `http://arxiv.org/abs/cs.PL/0105015`
112. van Hoeve, W.J., Pesant, G., Rousseau, L.M.: On global warming: Flow-based soft global constraints. *Journal of Heuristics 12*(4), 347–373 (2006)
113. Vanhoucke, M., Maenhout, B.: NSPLib – a nurse scheduling problem library: A tool to evaluate (meta-)heuristic procedures. In: *Proc. ORAHS2005* (2005)
114. Vanhoucke, M., Demeulemeester, E., Herroelen, W.: An exact procedure for the resource-constrained weighted earliness–tardiness project scheduling problem. *Annals of Operations Research 102*(1), 179–196 (2001)
115. Wang, W., Søndergaard, H., Stuckey, P.J.: A bit-vector solver with word-level propagation. In: C.G. Quimper (ed.) *Integration of AI and OR Techniques in Constraint Programming: 13th International Conference, CPAIOR 2016, Banff, AB, Canada, May 29 - June 1, 2016, Proceedings*, pp. 374–391. Springer International Publishing, Cham (2016)